

TECHNICAL UNIVERSITY OF CRETE  
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING



# CLONE: Cloud Ontology Editor

by

Alexandros G. Preventis

A thesis submitted in partial fulfillment of the requirements for the  
degree of  
Master of Science  
in Electrical and Computer Engineering

Chania, October 2020

## **Abstract**

Ontology development is a collaborative process that can involve several persons participating in different ways. The evolution of Web Services technology has facilitated collaboration on the Web, providing the means for simultaneous editing, tracking of changes and storing files on the cloud. Ontology development teams could greatly benefit from these collaboration features, that until now have been applied mainly to document processing. In this work, we introduce CLONE, a light-weight, Web based ontology editor that provides a real-time collaborative environment for creating and editing RDF and OWL ontologies. CLONE is designed using a component-based, service-oriented architecture taking advantages of the easy extensibility and scalability features of this approach. CLONE provides all the essential features of stand-alone ontology editors, as well as significant collaboration features, including simultaneous editing, change history, team conversations and role-based access-control mechanisms.

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Problem Definition . . . . .	8
1.2	Proposed Solution . . . . .	9
1.3	Thesis Outline . . . . .	10
<b>2</b>	<b>Background and Related Work</b>	<b>11</b>
2.1	Semantic Web . . . . .	11
2.2	RDF & RDFS . . . . .	12
2.3	OWL . . . . .	12
2.4	OWL API . . . . .	14
2.5	Ontology Engineering . . . . .	14
2.6	Collaborative Ontology Editors . . . . .	15
2.6.1	OntoWiki . . . . .	16
2.6.2	NeOn Toolkit . . . . .	16
2.6.3	Web Protégé . . . . .	17
2.6.4	OntoStudio . . . . .	18
2.6.5	Overview . . . . .	18
<b>3</b>	<b>Requirements Analysis</b>	<b>21</b>
3.1	User Roles . . . . .	23
3.2	Use Cases . . . . .	23
3.2.1	Ontology Management . . . . .	23
3.2.2	Ontology Editing . . . . .	24
3.3	Activities . . . . .	26
3.3.1	Ontology Management . . . . .	26
3.3.2	Ontology Editing . . . . .	36
<b>4</b>	<b>Architecture</b>	<b>41</b>
4.1	Challenges and Decisions . . . . .	41
4.2	Architectural Layers . . . . .	42
4.2.1	Presentation Layer . . . . .	42
4.2.2	Application Layer . . . . .	44
4.2.3	Data Access Layer . . . . .	44

4.3	Software Components . . . . .	44
4.3.1	Web UI Component . . . . .	44
4.3.2	Transformation Component . . . . .	44
4.3.3	Authorization Component . . . . .	46
4.3.4	Clone Core Component . . . . .	46
4.3.5	Ontology Editing Service . . . . .	48
4.3.6	Ontology Repository . . . . .	49
4.3.7	Reasoning Service . . . . .	52
4.4	Overall Architecture . . . . .	54
<b>5</b>	<b>Implementation</b>	<b>55</b>
5.1	Deployment . . . . .	55
5.2	User Interface . . . . .	56
5.2.1	Sign-in . . . . .	56
5.2.2	Home page . . . . .	56
5.2.3	Ontology Editor . . . . .	62
5.2.4	Ontology Tabs . . . . .	66
5.2.4.1	The Ontology tab . . . . .	66
5.2.4.2	Classes . . . . .	67
5.2.4.3	Object Properties . . . . .	68
5.2.4.4	Data Properties . . . . .	69
5.2.4.5	Annotation Properties . . . . .	71
5.2.4.6	Individuals . . . . .	71
5.2.4.7	Datatypes . . . . .	72
<b>6</b>	<b>Conclusions and Future Work</b>	<b>74</b>
	<b>Appendices</b>	<b>75</b>
<b>A</b>	<b>Database Schemata</b>	<b>76</b>
A.1	Clone Core Component's SQL Database . . . . .	76

# List of Figures

2.1	Simple RDF graph . . . . .	13
3.1	Use Cases: Ontology Management . . . . .	25
3.2	Use Cases: Ontology Editing . . . . .	27
3.3	Create new Ontology . . . . .	30
3.4	Import Ontology . . . . .	30
3.5	Upload Ontology . . . . .	30
3.6	Download Ontology . . . . .	31
3.7	Leave Ontology . . . . .	31
3.8	Delete Ontology . . . . .	31
3.9	Activity diagram: Add ontology user . . . . .	32
3.10	Activity diagram: Create ontology version . . . . .	33
3.11	Activity diagram: Manage ontology versions . . . . .	34
3.12	Activity diagram: Edit ontology information . . . . .	35
3.13	Activity diagram: View Ontology . . . . .	38
3.14	Activity diagram: Edit Ontology . . . . .	39
3.15	Activity diagram: Apply Reasoning . . . . .	40
4.1	Architectural Layers . . . . .	43
4.2	Architectural Layers with Components . . . . .	45
4.3	Architecture of the Clone Core Component . . . . .	47
4.4	Architecture of the Ontology Editing Service . . . . .	50
4.5	Architecture of the Ontology Repository Service . . . . .	52
4.6	Architecture of the Reasoning Service . . . . .	53
4.7	Overall Architecture . . . . .	54
5.1	The nodes where CLONE's artifacts have been deployed. . .	56
5.2	Sign-in . . . . .	57
5.3	Home page . . . . .	57
5.4	Create new ontology . . . . .	58
5.5	Insert ontology from URL . . . . .	59
5.6	Insert ontology from URL . . . . .	59
5.7	Ontology menu . . . . .	60
5.8	Edit ontology's basic information . . . . .	61

5.9	Manage user roles . . . . .	62
5.10	Manage ontology versions . . . . .	63
5.11	Ontology Editor . . . . .	63
5.12	Reasoning result . . . . .	64
5.13	Ontology history and conversation . . . . .	65
5.14	Ontology tab . . . . .	66
5.15	Classes tab . . . . .	67
5.16	Object Properties tab . . . . .	70
5.17	Data Properties tab . . . . .	70
5.18	Individuals tab . . . . .	72
5.19	Individuals tab . . . . .	73
A.1	Clone Core Component's database schema . . . . .	78

# List of Tables

2.1	Overview of Collaborative Ontology Editors . . . . .	20
-----	--	----

# Chapter 1

## Introduction

In the past few years the World Wide Web has evolved from a medium for displaying information, to an environment where people can communicate, work, collaborate and exchange content. This new “form” of the Web, known as *Web 2.0*, mostly consists of content generated by users. The next step in Web’s evolution is called the Semantic Web (also referred to as *Web 3.0* or *Web of Data*). The Semantic Web provides a common framework that makes data machine-readable by enhancing it with metadata. This metadata provide information on what the data represents and how it should be processed. A set of metadata that describes the concepts of a specific domain can be represented by means of an ontology.

An ontology (also known as *vocabulary*) describes the concepts that describe a domain of interest and also the relationships that hold between those concepts. Ontologies enable sharing the structure of information, so that it can be processed by people or agents. For example, in the case of public transportation, companies sharing and publishing an ontology, would allow agents to extract and aggregate data making it possible to answer queries about every single route (e.g., pricing, duration etc.). Ontologies also make knowledge reusable, allowing it to be used by other people, or applications, in order to describe a different (or more specific) domain of interest. For example, an ontology about *edible things* could be used and extended in order to describe the domain of *fruits*.

There are several data models for expressing ontologies (ontology languages), the most prevalent of which are the Resource Description Framework (RDF) and its more expressive descendant, the Web Ontology Language (OWL). RDF and OWL are designed to be machine-readable, thus it is difficult for developers to use them directly to create ontologies. *Ontology Editors* provide the means of creating, editing and manipulating ontologies, by offering user interface, hiding the syntax complexity of an ontology language and taking over the ontology language generation in the background. There are several ontology editors available, each one with different features



and advantages. Developers can choose an editor according to their needs. Most of the editors, though, are stand-alone applications that are executed in native environments. They are well-suited for sole ontology development, without providing any means for collaboration.

## 1.1 Problem Definition

Building an ontology usually requires more than a skilled developer and an ontology editor. In fact it is a collaborative task in which, apart from the developers, there are several other participants who provide knowledge and information on the domain of interest. Braun et al. [1] describe this process as an informal learning process, called “Ontology Maturing,” that can only be fulfilled through collaboration.

To enable collaboration, participants of the ontology creation process need to:

- have direct and ubiquitous access to the ontology, enabling them to access it at any time from any place
- be in constant and real-time communication with each other in order to discuss and propose changes, resolve differences and provide help to each other
- be able to edit the ontology simultaneously and view changes from all editors of the ontology when they occurred,
- keep a track of ontology changes, which includes a detailed description on the changes, including the editor of each change, the type of the change and the time it occurred (e.g., *John on 2015/07/01 14:54* edited the superclasses of class *Animals*),

These requirements (along with others described in Chapter 3), until now have been partially fulfilled by sending e-mails (for communication and sharing ontology files), using social media applications and file-sharing applications. This sort of communication and sharing can seriously obstruct the development process, as the members cannot participate in it simultaneously, but they have to edit the ontology one by one, locally, and then pass it to the others.

The ideal solution to overcome this obstacle would be ontology editors that enable manipulation of RDF and OWL ontologies and also provide the aforementioned collaboration features. Existing collaborative editors mostly focus on enhancing document content with semantics. Only few of them fully support editing of the structure of ontologies and even fewer provide editing functionalities in full extend of the W3C specifications<sup>1</sup>.

---

<sup>1</sup><http://www.w3.org/TR/owl2-syntax/>

## 1.2 Proposed Solution

We introduce *CLONE* (Cloud Ontology Editor), a lightweight, Web-based ontology editor that provides a real-time collaborative environment for creating and editing RDF and OWL ontologies. It's friendly and easy-to-use interface allows users to create ontologies without being familiar with the peculiarities (e.g. syntax) of the underlying ontology representation language. CLONE runs in the cloud and requires no software installation on the user's machine. Users can access and edit their ontologies from any place or machine, just by using a Web-browser.

CLONE has been designed as a component-based, service-oriented distributed system. It is composed of autonomous, reusable components that operate as Web services running in the cloud and communicate with each other through RESTful interfaces. CLONE's architecture allows for (a) load-balancing, as heavy work is distributed between the components, (b) extensibility, as new component that provide different services can easily be added to the system and (c) scalability, as nodes that have to do much "heavy lifting" can easily be replaced with others that utilize more physical resources.

CLONE provides all the essential features of stand-alone ontology editors, but also provides significant collaboration features. These include the following:

- Ontology creation. Users can create ontologies from scratch or by uploading an ontology file. They can also import ontologies directly from the Web.
- Private repositories. Users maintain a private ontology collection which includes the ontologies they have created or uploaded. Collections can be modified by their owners at any time.
- Full support of OWL 2 axioms, as they are defined in OWL 2 structural specification, allowing users to exploit OWL's expressiveness to the fullest extent, enabling them to create more elaborate and detailed ontologies.
- Reasoning support. Users can apply reasoning over their ontologies to check consistency or even to view the new, inferred ontology.
- Comprehensive error-recovery mechanisms, including flow control buttons (Undo/Redo) that allow users to undo up to 50 changes on the ontology and version control mechanism that allows users to return to previous states of the ontology.
- Team editing. Ontology owners can invite other people to collaborate in the engineering process.

- Access-control mechanisms. Ontology owners can assign roles to the users they invite. There are three different user-roles that can be assigned: viewer, editor or administrator, each providing additional access-rights compared to the role before it in the list.
- Simultaneous editing. All users involved in the editing process, can make changes on the ontology at the same time. Changes made by one user will instantly appear to all other users editing the ontology.
- Change tracking. Information for each change made on the ontology is recorded and becomes available to all members of the editing team.
- Team conversation. The editor provides a real-time conversation mechanism facilitating communication between the editing team.

The prototype is hosted on the Intelligent System Laboratory's cloud infrastructure. A public instance of CLONE is available on the Web<sup>2</sup>.

### 1.3 Thesis Outline

Related work in the field of ontology engineering is discussed in Chapter 2. This includes a description of the Semantic Web and related technologies, work on the ontology engineering process and presentation of existing collaborative ontology editors and their features. A requirement analysis for collaborative ontology editors is presented in Chapter 3. In Chapter 4 there is discussion about the architecture of the overall system and its components. Details on CLONE's implementation are presented and discussed in Chapter 5. Finally, conclusions and issues for future work are discussed in Chapter 6.

---

<sup>2</sup><http://www.intelligence.tuc.gr/clone/>

## Chapter 2

# Background and Related Work

### 2.1 Semantic Web

The Semantic Web [2] (also known as Web 3.0 or Web of data) is the evolution of the World Wide Web that enables content to be linked, shared beyond the boundaries of applications and reused. This practically means that data published by one application, can be understood, processed and used by other applications to produce useful results. In that end, the activities that have been initiated by the W3C (previously the Semantic Web Activity<sup>1</sup> and now the Data Activity<sup>2</sup>), have developed technologies that enable people to create data stores on the Web, build vocabularies, and write rules for handling data.

The main areas of Semantic Web are the following:

- **Linked Data** focuses on making data, and relationships between data, on the Web available in a standard format, reachable and manageable by Semantic Web tools. W3C's standardizations in that area, among others, include RDF [3] and OWL [4].
- **Vocabularies** provide a formal representation of concepts that are used to describe a particular domain of interest. Vocabularies are used to classify the terms that can be used in a particular application, characterize possible relationships, and define possible constraints on using those terms. W3C's standardizations in that area include RDF and RDF Schemas, Simple Knowledge Organization System (SKOS) [5], OWL and Rule Interchange Format (RIF) [6].
- **Queries**, meaning the technologies that provide the means for retrieving information from the Web of Data. The standard for querying in

---

<sup>1</sup><http://www.w3.org/2001/sw/>

<sup>2</sup><http://www.w3.org/2013/data/>

the Semantic Web is SPARQL [7], a query language for retrieving data stored in RDF format.

- **Inferences** provide the means for discovering new relationships or possible inconsistencies between data. Inferences improve the quality of data integration on the Web, by automatically analyzing the content of the data, or managing knowledge on the Web in general. Inferences are generated by the use of software tools called Reasoners.
- **Vertical Applications** is a term for describing generic application areas, specific communities, etc, that explore how Semantic Web technologies can be used to help their operations, improve their efficiencies, provide better user experiences, etc.

## 2.2 RDF & RDFS

One of Semantic Web's objectives is to make data machine readable. To do so, the need arises for formal ways to describe concepts of application domains and the relationships that hold between them. This formal representation of knowledge is called an ontology.

There are several data formats for encoding ontologies, among them, the Resource Description Framework (RDF) [3], that is a W3C standardization. RDF is based on the idea that each resource on the Web is identified by a unique identifier (called Uniform Resource Identifier) and can be described in terms of properties and property values. Statements about resources in RDF have the form of subject–predicate–object expressions, that are called *triples*. For example the statement “John Smith lives in London” can be decomposed as follows: “John Smith” is the subject, “London” is the object and “lives in” is the predicate. This way, statements in RDF can be represented as graphs with the nodes being the resources and arcs being the properties. Figure 2.1 illustrates a simple RDF graph.

RDF is suitable for describing ontologies, but it provides limited expressiveness. To overcome this limitation, an extension of RDF was introduced, called RDF Schema (RDFS) [8]. RDFS is a general-purpose language for representing RDF vocabularies. It provides a set of constructs (classes, properties), built on the limited vocabulary of RDF, that can be used for the description of ontologies. RDFS is also a W3C recommendation<sup>3</sup>.

## 2.3 OWL

The Web Ontology Language (OWL) [4] is a semantic markup language for publishing and sharing ontologies on the World Wide Web. It is developed

---

<sup>3</sup><http://www.w3.org/TR/rdf-schema/>

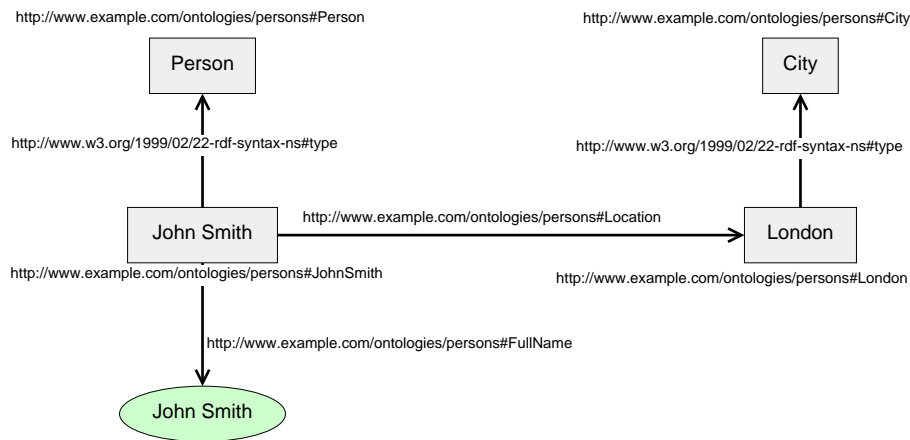


Figure 2.1: Simple RDF graph representation of the statement “John Smith is a person that lives in London.”

as a vocabulary extension of RDF and is derived from the DAML+OIL Web Ontology Language [9]. OWL is intended to provide a language that can be used to describe the classes and relations between them that are inherent in Web documents and applications. Any OWL ontology can also be described as an RDF graph.

OWL provides three increasingly expressive sub-languages:

1. **OWL Lite** supports classification hierarchy and simple constraints.
2. **OWL DL** supports maximum expressiveness without losing computational completeness and decidability.
3. **OWL Full** provides maximum expressiveness and the syntactic freedom of RDF, but without computational guarantees.

Each of these sublanguages is an extension of its simpler predecessor.

OWL is part of W3C’s Semantic Web technology stack. In October 2007 a new W3C working group was started to extend OWL with several new features. This new version was called OWL 2 [10]. Though OWL 2 adds new functionality with respect to OWL 1, it should be noted that it provides complete backwards compatibility, to all intents and purposes. This means that all OWL 1 ontologies are valid OWL 2 ontologies.

OWL 2 provides three more sub-languages, called *Profiles*, that offer important computational or implementational advantages in particular application scenarios. Each profile is a subset of the structural elements that are provided by OWL 2 DL and all are more restrictive than OWL 2 DL. The profiles are:

1. **OWL 2 EL** which enables polynomial time algorithms for all the standard reasoning tasks.

2. **OWL 2 QL** which enables conjunctive queries to be answered in LogSpace using standard relational database technology.
3. **OWL 2 RL** which enables the implementation of polynomial time reasoning algorithms using rule-extended database technologies operating directly on RDF triples.

## 2.4 OWL API

The OWL API [11] is high level application programming interface (API), written in Java, for working with OWL ontologies. It is fully compliant with W3C’s OWL 2 structural specification. OWL API supports ontology management, ontology change, ontology parsing and rendering, data structure storage and reasoner interfaces. The latest releases of the OWL API also provide validation methods for the OWL 2 ontology profiles.

## 2.5 Ontology Engineering

Ontology engineering refers to “the set of activities that concern the ontology development process, the ontology life cycle, the principles, methods and methodologies for building ontologies, and the tool suites and languages that support them” [12].

In the past, the methodologies for ontology engineering had a centralized approach that usually involved a small group of people that consisted of domain experts that provided the knowledge to be modeled and knowledge engineers that structured and formalized the ontology. These methodologies regard ontologies as something stable that, since developed, is never changed.

As ontologies’ popularity increased and they started to be used in more and more applications, the number of people involved in ontology engineering increased as well. Ontologies grew in size, with some of them containing thousands or even millions of concepts (e.g., NCI Thesaurus<sup>4</sup> contains 110,000 concepts and 400,000 inter-concept relationships). Effective development of such ontologies from one person or a small group of people is impossible. The need for more people to participate in the ontology engineering process became apparent and imminent. Ontology engineering became a social, collaborative, evolving process that involves a geographically dispersed community—ontology engineers, domain experts and ontology users—with different knowledge and expertise [13, 14].

The emerge of ontology editors enabled more people to actively participate in the ontology engineering process, since explicit knowledge of ontology

---

<sup>4</sup><https://ncit.nci.nih.gov/ncitbrowser/>

languages was no more required. Ontology editing could be done by any participant, separately, just by using a desktop application. The difficulties that arose with this approach, concern the synchronization of multiple different ontology versions that are created by each of the editors, as well as the lack of means to facilitate communication between participating editors. The problem of synchronization was usually addressed (a) either by all proposed changes being approved or rejected by a “lead editor” who eventually publishes a formal version of the ontology, or (b) by the use of version control systems, which mostly rely on textual differences between files. The communication problems were partially tackled with the use of e-mails, instant messaging and discussion forums.

All these problems have been definitely addressed with the appearance of *Collaborative ontology editors*, which are specialized software tools that provide, apart from ontology editing features, some of the desired features that facilitate collaboration during the ontology engineering process. Collaborative ontology editors come either in a form of extension on already existing ontology editors [15, 16], or as new ontology editors that are explicitly designed to support collaboration [17]. The adoption of Collaborative editing tools led to the development of new decentralized ontology engineering methodologies. Simperl and Luczak-Rösch [18], in their work, present a detailed survey on collaborative ontology editing tools as well as the ontology engineering methodologies that have been developed over the years.

Collaborative ontology editors well facilitate the manual ontology development process enabling groups of people to participate actively. The volume of RDF data that is being published on the Web, though, is so huge that even with collaboration, it is practically infeasible to be managed by people. The next step for ontology development is creating tools that integrate human and computational intelligence. In the future, ontology engineering methodologies orient from a tool-assisted, but primarily manual engineering process to a data-driven approach in which human involvement is optimally leveraged for the resolution of those issues that can not be feasibly automatized.

## 2.6 Collaborative Ontology Editors

The main challenge in collaborative ontology engineering is choosing the right tool that offers multiple-user access at the knowledge level, adequate version management, as well as communication mechanisms that enable effective collaboration between geographically distributed participants.

In the past few years, many ontology editors have been developed that, partially or fully, satisfy these requirements. In the scope of this work, we present some of the ontology editing tools that provide collaboration mechanisms and are still under active development or maintenance.



### 2.6.1 OntoWiki

OntoWiki [19] is a Semantic Wiki application for managing structured data in a collaborative, Web-based environment. It allows users to navigate through RDF knowledge bases and view Linked Data in a wiki-like form<sup>5</sup>. Users can create different views on data, such as tables, calendars and maps. RDF content can be edited in-line on the pages that it is displayed. It can also be exported in various serialization formats, suitable for machine-consumption. OntoWiki includes a version control mechanism that allows each page to roll back to previous states. Finally, reasoning and consistency checking can be applied with the use of DL-Learner plug-in for OWL.

OntoWiki, like most Wiki based ontology editors where any user can add or edit content, trades part of the expressiveness of the ontology with ease-of-use, resulting to limited expressiveness, compared to other ontology editors. Moreover, it does not provide role management, allowing any registered user to modify the Ontology data. Lastly, OntoWiki does not provide any explicit communication mechanism between the collaborating parties.

### 2.6.2 NeOn Toolkit

NeOn Toolkit [20] is based on the Eclipse<sup>6</sup> platform and provides several plug-ins, each one providing different functionality. The standard platform offers common functionalities such as ontology browsing, editing, import and export in various formats (OWL, F-Logic, UML). Additional functionalities, that are essential for ontology engineering (i.e., ontology visualization, reasoning, collaboration), are only provided by the use of plug-ins.

The two plug-ins that enhance NeOn Toolkit with collaboration functionalities are:

- WikiFactory, that enables automatic creation of semantic wiki-based Web sites, allowing for dynamic content management and content synchronization with the underlying OWL Ontology.
- Cicero, that enables asynchronous discussions on aspects related to ontology-engineering and also supports decision making. Cicero is based on the DILIGENT knowledge process, a methodology for collaborative ontology engineering [13].

These two plug-ins provide limited collocation features to NeOn. Simultaneous ontology editing and central version control are not supported. Instead, every user has to keep and work on a local copy of the ontology and need to invoke the OWLdiff plug-in in order to locate the differences with a central copy of the ontology and only apply the differences. Finally, NeOn

---

<sup>5</sup>Wiki is a website or database developed collaboratively by a community of users, allowing any user to add and edit content

<sup>6</sup><https://eclipse.org/>

Toolkit does not provide sufficient role management in ontology editing, as any contributor can apply any changes.

### 2.6.3 Web Protégé

Web Protégé [21] is a collaborative Web-based platform that supports ontology editing and knowledge acquisition. It started as a Web front-end for the Collaborative Protégé server, a plug-in that had been created for the 3<sup>rd</sup> version of the stand-alone Protégé platform, that provided collaboration capabilities. Since this plug-in has not been supported by the next versions of Protégé, the Web platform is the only collaboration editor that is actively supported by the Protégé community.

Web Protégé is one of the most comprehensive tools for collaborative ontology engineering that is currently available. It provides most of the desired collaborative features. Most specifically, it provides:

- highly adaptable and customizable user interface. The users can add tabs and views and adjust the layout in many ways to better suit their needs.
- multiple-user access to the ontology and simultaneous editing. Any changes made by one user are immediately visible to others.
- communication mechanisms, in form of a chat channel for direct communication or discussion threads that users have created.
- central ontology repository with access control mechanisms. Ontology are accessible only to those that have the appropriate permissions.
- various formats for ontology import/export (RDF/XML, OWL/XML, Turtle, etc.).
- 3 different roles for the users (editor that can make changes to the ontology, viewers that can only view the ontology and commentators that can participate in the discussions).
- annotations in a form of ontology (Changes and Annotations Ontology – ChAO), this way allowing for annotating the changes made on the ontology and also commenting on the ontology engineering process itself.
- change history, in the form of revisions, each representing atomic user actions.
- versioning, allowing to download the ontology state in a specific revision.
- graph visualization of the ontology concepts and their interrelations.

Web Protégé is very similar to the stand-alone Protégé platform in many ways. It is surpassed, though, by the stand-alone version in aspects of expressiveness as it does not support all OWL2 axioms. It also does not provide any mechanisms for consistency checking or reasoning.

#### 2.6.4 OntoStudio

OntoStudio [16] is a commercial ontology engineering platform created by Semafora<sup>7</sup> that originates from the NeOn Project. Thus, it is also based on the Eclipse platform and it provides many plug-ins that offer various functionalities:

- It allows creating, browsing, maintaining and managing ontologies.
- It uses a central repository where the ontologies can be stored and edited by multiple-users simultaneously, this way allowing collaborative development.
- It supports multiple import/export formats (OWL, RDF, F-Logic, UML).
- It supports reasoning and consistency checking.
- It allows users create maps between the concepts of their ontology and either other ontologies, or databases, through a graphical editor that is provided by the OntoMap plug-in. After the mappings have been applied, the user can query on these data-sources using the concepts of their ontology.
- It provides a graphical and a textual rule editor, that enables users to enrich their ontologies with additional semantics and possibilities.

OntoStudio offers a Web interface that can be used for applying very simple changes on ontologies, such as editing classes or other entities.

OntoStudio provides most of the essential functionalities for ontology engineering, plus some additional functionalities that are not found in other ontology editors (i.e., ontology mapping). It targets, though, users that are familiar with ontology engineering and can be considered experts. Its interface is not suitable for users that have no previous experience in knowledge engineering and representation and it cannot easily be used by them.

#### 2.6.5 Overview

Table 2.1 presents an overview of the ontology editors that have been discussed, along with the features that they provide. It is clear that, though

---

<sup>7</sup><http://www.semafora-systems.com/en/>

support of collaborative ontology engineering has been provided by some editors, no editor provides full collaboration support in all the stages of ontology development.

Table 2.1: Overview of Collaborative Ontology Editors

Editor	Multi-user access & editing	Versioning	Consistency checking	Role management	Communication	Cloud implementation & SOA design
OntoWiki	Wiki-based access and editing, concurrency control at page level.	Version control in page level. Differences between versions. Rollback.	Provided by plug-in.	Contributors.	No communication support.	No cloud implementation. Operates as a Web application.
NeOn Toolkit	Not supported. Local access and editing.	No version-control.	Consistency checking via plug-ins.	Provided via Cicero plug-in. Contributors.	Enabled by Cicero plug-in. Conforms to the DILIGENT methodology.	No cloud implementation. Operates as a desktop application.
Web Protégé	Simultaneous access and editing. Changes immediately seen by other users. Changes automatically saved to central repository.	Revisions automatically created on each change. Ability to rollback to any revision.	No consistency checking or reasoning support.	Editors, Commentators and Viewers	Instant messaging, threaded discussions, annotations of changes	Cloud implementation. No SOA design. Operates as a Web application.
OntoStudio	Simultaneous access and editing. Requires connection of the stand-alone application to the Collaboration Server to enable concurrency control.	No version control.	Consistency checking and reasoning support.	Contributors.	No communication support.	No cloud implementation. Operates as a desktop application.
CLONE	Simultaneous access and editing. Changes immediately seen by other users. Changes automatically saved to central repository.	Revisions created by users. Ability to rollback to any revision.	Consistency checking and reasoning support.	Administrators, Editors and Viewers	Instant messaging.	SOA design and Cloud implementation.

## Chapter 3

# Requirements Analysis

Collaborative ontology editing is a very well-researched topic. The works by Tudorache et al. [15], and Simperl et al. [18] highlight user requirements and the functionality that collaborative ontology editors should support. Building upon their work, CLONE has been designed to support the following features:

**Shared Access:** All users have access to the same resources. Collaborative ontology editors should allow all users participating in the ontology engineering process to access the same file, at the same time.

**Access Control:** Access to the ontology should be controlled by the ontology administrators. Administrators are entitled to grant access to users (i.e. developers or editors). Users may have been granted access to the whole ontology or only to specific parts of it. However, they are allowed to view the entire ontology. This particular feature is important in the case of large ontologies, where the development of different ontology parts may be assigned to different parties.

**Role Management:** In collaborative ontology development, not all participants contribute in the same way. Domain experts contribute by providing knowledge on the domain of interest, but they rarely get involved into the development process. Thus, they should be able to browse the ontology and communicate with the developers to propose possible modifications, but they aren't always allowed to modify the ontology themselves. On the other hand, developers are allowed to modify the entities of the ontology and also participate in conversations with other users and consult the domain experts. Finally, there are users that are allowed to administrate the ontology: (a) to manage the ontology project, (b) to manage the users that have access to the ontology, as well as their permissions, (c) to be able to browse and edit any part of the ontology.

**Versioning:** Users can create and maintain versions of an ontology that is under-development. These versions can be stored in different locations (i.e. the users can maintain local copies of the ontology in their private computers), or they can be stored centrally in the system’s database. Versions stored in the system can be used as restoration points of the ontology.

**Error Recovery:** The editing tool should enable the users to recover from errors. This can be done by preventing errors (e.g., prompt users to confirm that they want to complete some actions, otherwise enable them to cancel these actions), by implementing backwards error recovery functions (e.g., provide “Undo” and “Redo” buttons), or by enabling users to roll-back the system to a previous working state such as a previous ontology version. Among others, administrators can choose to roll back part or the entire ontology to a previous stored version.

**Communication:** One of the most essential features of collaborative ontology development is enabling communication between the participants, allowing them easily to share knowledge and ideas with each other. Communication can be instant (like a chat) allowing users to communicate in real time, or in a form of threaded discussions.

**Expressiveness:** OWL 2, is the most expressive of Semantic Web’s ontology languages. It supports complex expressions such as defined classes, complex property restrictions, negative property assertions etc. A collaborative ontology editor should provide full support of the expressiveness that OWL 2 offers, allowing ontology developers to use this expressiveness to its full extend. This is a feature that currently is not supported by any of the existing collaborative ontology editors. They mostly allow developers to perform simple ontology actions, such as modify class hierarchy or create object and data property assertions.

**Consistency Checking:** In collaborative ontology development, consistency checking of the ontology under development, is even more crucial. Enabling multiple users to simultaneously edit the same ontology file and its included entities, may lead to many inconsistencies and errors. It is crucial that a collaborative ontology editor provides built-in consistency checking mechanisms that allow users to check the ontology at any time.

CLONE’s functionality fulfills all the aforementioned requirements. In section 3.1 we define CLONE’s user roles. Sections 3.2 and 3.3 describe the use cases and the activities that can be performed, respectively, from the viewpoint of two functional aspects: *Ontology Management* and *Ontology Editing*.

## 3.1 User Roles

In order to use CLONE users will have to register to the system. After their registration they can become part of an editing team, either as ontology creators or as invitees to other user's ontologies. Each editing-team member is assigned a role, that defines the actions they can perform in the ontology. CLONE distinguishes between three user roles, each one with different privileges and permissions.

**Viewer:** A viewer can view the ontology and all its entities (Classes, Individuals, etc.) but is not allowed to make any modifications. Viewers can also view the change history of the ontology and participate in its conversation.

**Editor:** Editors can perform all actions that viewers can. Moreover, they can modify the ontology entities.

**Administrator:** Ontology administrators have unrestricted access to the ontology. They can manage the ontology (i.e. edit/delete it, create new versions) and its editing team by adding/removing users or changing the user roles. They also can modify the ontology entities. Ontology creators are automatically assigned the “administrator” role.

## 3.2 Use Cases

CLONE's user actions can be separated into two functional aspects: *Ontology Management* and *Ontology Editing*. Ontology Management includes the set of actions that users can perform on ontology files, while Ontology Editing includes the set of actions that users perform on the Entities of an ontology.

In this Section we discuss the use cases of CLONE from the scope of these two aspects.

### 3.2.1 Ontology Management

The user actions for Ontology Management are:

- (i) Creation of a new, empty ontology.
- (ii) Insert an ontology from a Web location just by inserting its URL.
- (iii) Upload an ontology file from the local machine.
- (iv) View the details of an ontology, including the ontology's name, the URL where the ontology file is stored, a short description of the ontology, the date that it was created and the username of its creator.



- (v) Download an ontology to the local machine.
- (vi) Leave the editing team of an ontology. By doing so, the user renounces any permissions that they had been granted on the ontology. This does not apply on ontology creators who cannot leave its editing team.
- (vii) Manage user roles on a specific ontology. Ontology creators can allow other users to view, edit or manage their ontologies, by assigning roles, as described in 3.1.
- (viii) Delete user's role, removing any permissions that had been granted to them.
- (ix) Create new versions of an ontology. A version is a snapshot of the current state of the ontology that can serve as restoration point. Administrators can create ontology versions at any time.
- (x) Change the active version of an ontology. Administrators can swap between ontology versions, allowing them to role-back the ontology in case of errors.
- (xi) Delete ontology versions. In the case that there are versions of the ontology that have become obsolete (e.g., they have been created in the early stages of the development process) administrators can delete them.
- (xii) View version details, such as the date and time that the version has been created and its creator.
- (xiii) Edit ontology name.
- (xiv) Edit ontology's description.

Figure 3.1 illustrates the aforementioned actions, associated with the actors (different users) of CLONE.

### 3.2.2 Ontology Editing

The actions for Ontology Editing are:

- (i) View Ontology. This use-case includes the actions for browsing the ontology and viewing its classes, object properties, data properties and individuals.
- (ii) Download an instance of the current ontology.
- (iii) Apply Reasoning on the ontology. All ontology users can invoke the Reasoning Service allowing them to check for possible inconsistencies in the ontology. After the reasoning is completed, the user is informed about any inconsistencies that have been found.

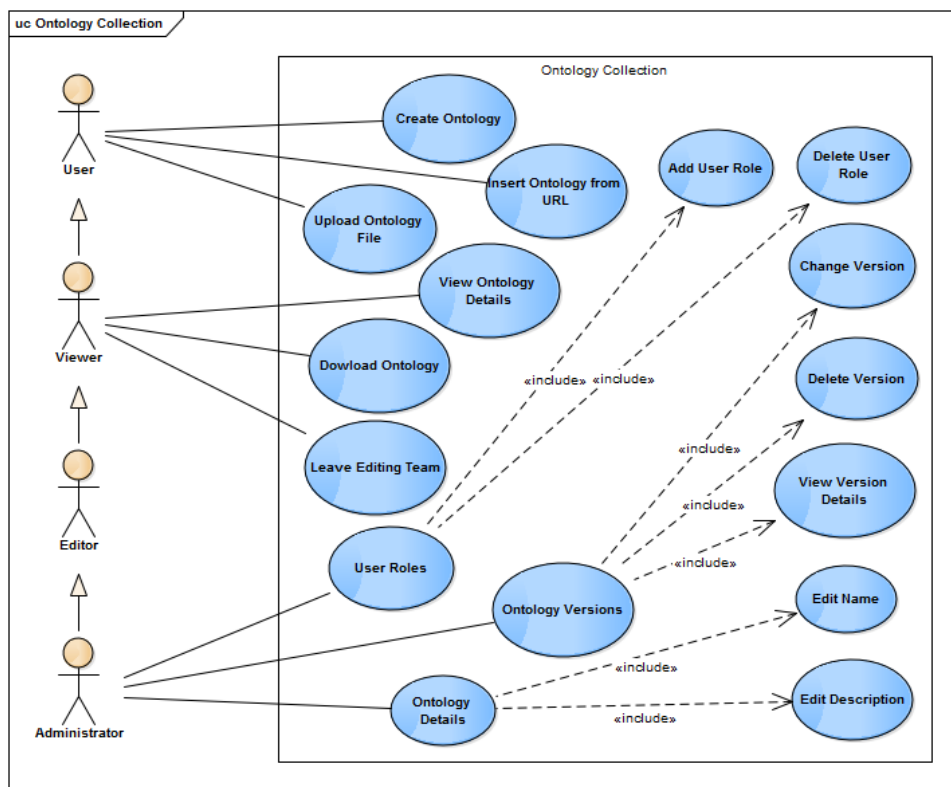


Figure 3.1: Use Cases: Ontology Management.

- (iv) View Inferred ontology. Following the reasoning action, provided that the ontology is consistent, the user can choose to view the Inferred ontology<sup>1</sup>.
- (v) Download Inferred ontology. Though users are not able to edit the Inferred ontology version, they can download it to their local machine.
- (vi) Participate in conversation. The ontology editing environment provides a chat service to facilitate the communication between the editing team. Through the chat view, users can also see a list with the members of the editing team that are currently active.
- (vii) View Ontology changes. CLONE keeps a history with every change that has taken place in the ontology from the moment of its creation. Users are able to view these changes as well as who has made them.
- (viii) Create new ontology version. Editors and Administrators can take snapshots from the ontology. These snapshots can be used from the administrators as restoration points.
- (ix) Edit ontology. This use-case includes all the actions that can, in any way, modify the *axioms* of the ontology. CLONE provides full support of the OWL2 axioms, as they are described in “OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax”<sup>2</sup>.

Figure 3.2 illustrates these use-cases, associated with their actors. The users that act as Viewers can only perform actions that do not change the content of the ontology. Editors can edit the content. In the editing environment, Administrators can perform exactly the same actions as Editors.

### 3.3 Activities

In this Section, we describe the activities of the users in CLONE. Each activity is also described by means of UML diagram.

#### 3.3.1 Ontology Management

The “ontology management” activities refer to the use cases described in subsection 3.2.1. The user starts from their Home page, which is the page that the user is presented after they log-in to the system. The Home page is thoroughly described in subsection 5.2.2.

---

<sup>1</sup>Inferred ontology is the ontology that has been created by the reasoner, by including on the initial ontology all the inferred axioms that it has produced.

<sup>2</sup><https://www.w3.org/TR/owl2-syntax/#Axioms>

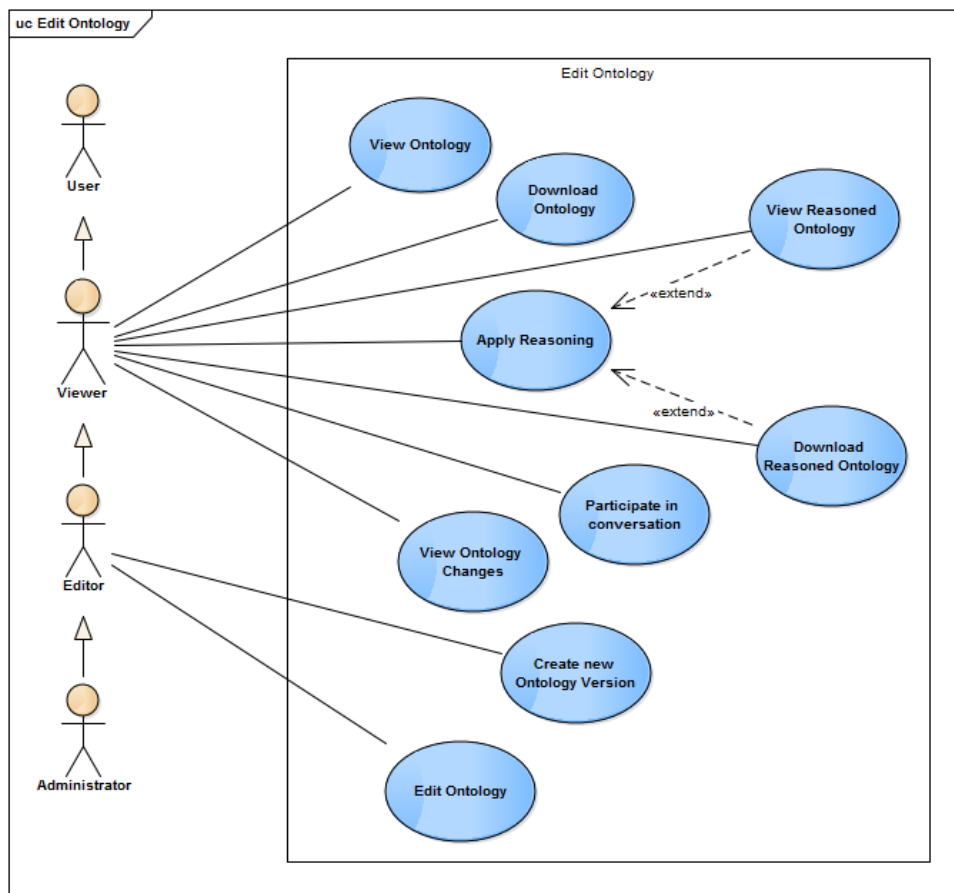


Figure 3.2: Use Cases: Ontology Editing.

**Create new Ontology:** The user clicks on the “Create Ontology” button. The “Create new Ontology” dialog window will show up, prompting the user to fill in the ontology name and optionally a description for the ontology. Then the user clicks on the “Save” button. A new blank ontology has been created and is available in the user’s home page. This activity covers the ontology management use case (i). Figure 3.3 illustrates the UML activity diagram.

**Import Ontology:** The user clicks on the “Insert from URL” button. The “Insert Ontology from URL” dialog window will show up, prompting the user to fill in the URL where the ontology is located. Then the user clicks on the “Insert” button. The ontology has been imported and is available in the user’s home page. This activity covers the ontology management use case (ii). Figure 3.4 illustrates the UML activity diagram.

**Upload Ontology:** The user clicks on the “Upload Ontology” button. The “Upload Ontology” dialog window will show up, prompting the user to fill in the ontology name and select the ontology file that will be uploaded. Then the user clicks on the “Upload” button. A new ontology has been created, from the uploaded ontology file, and is available in the user’s home page. This activity covers the ontology management use case (iii). Figure 3.5 illustrates the UML diagram.

**Download Ontology:** The user clicks on the ontology’s dropdown menu. The ontology file, containing the ontology exported in RDF/XML format, is downloaded on the user’s computer. This is the activity management use-case (v) and is illustrated as a UML activity diagram in Figure 3.6.

**Leave Ontology:** The user clicks on the ontology’s dropdown menu and selects the “Leave Ontology” option. The system then displays a dialog window asking the user to confirm that they want to leave the ontology. After the user confirms the action, they will be removed from the ontology editing team and any (view or edit) permissions that had been granted will be revoked. The corresponding ontology management use-case to this activity is (vi). Figure 3.7 illustrates the UML activity diagram.

**Delete Ontology:** The *Ontology Administrator* clicks on the ontology’s dropdown menu and selects “Delete Ontology”. A dialog window will appear asking the user to confirm the action. After confirming, the ontology and all related data (such as its editing team along with their roles, the change history, any stored versions, its conversation) is removed from the system. Any users viewing or editing the ontol-

ogy at the time of the removal will be redirected to their home page. Figure 3.8 illustrates the UML diagram of this activity.

**Manage User Roles:** The *Ontology Administrator* clicks on the ontology’s dropdown menu and selects “Manage Roles”. The “Manage Roles” dialog window will appear where the user add editing-team members and assign roles. Editing-team members can be specified by: (a) their username in CLONE or (b) the e-mail address that they used for registering to the system. Then, the administrator can select the role that will be assigned to the new team member. The role defaults to *Viewer* and can be changed to *Editor* or *Administrator*. Finally, the administrator clicks on the “Add” button and the new editing-team member is added. The specific ontology will be included to the new member’s home page. This activity corresponds to the ontology management use case (vii) and is illustrated in Figure 3.9.

**Create Ontology Version:** The user clicks on the ontology and opens it in the editing environment. Then he/she clicks on the “Create Version” button. A confirmation message will appear on the bottom of the screen notifying the user that the version has been created successfully. This activity is available to *Editors* and *Administrators*. It corresponds to the ontology management use case (ix). Figure 3.10 illustrates the UML activity diagram.

**Manage Ontology Versions:** The *Ontology Administrator* clicks on the ontology’s dropdown menu and select “Manage Versions”. The “Manage Versions” dialog window (described in subsection 5.2.2) will appear, providing information on the active ontology version (its creation date and the user who created it) and listing the available versions of the ontology (that are snapshots of the ontology taken by editors or administrators). In this window, the user can (a) view the current ontology version information, (b) change the active version of the ontology, by setting one of the version listed as “active” or (c) delete an ontology version. This activities are illustrated as a UML diagram in Figure 3.11. The use cases covered are: ontology management (x), (xi) and (xii).

**Edit Ontology Information:** The *Ontology Administrator* clicks on the ontology’s dropdown menu and selects “Edit Information”. The “Edit Ontology” dialog will appear where the user can change the name and/or the description of the ontology. Clicking “Save” will save the changes. Figure 3.12 illustrates the UML activity diagram. The use cases covered are: ontology management (xiii) and (xiv).

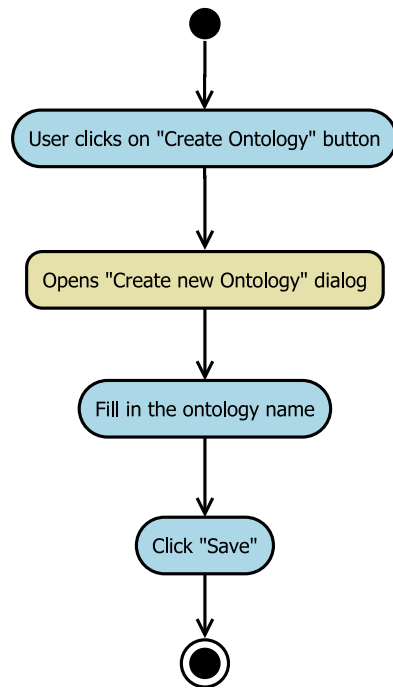


Figure 3.3: Create new Ontology

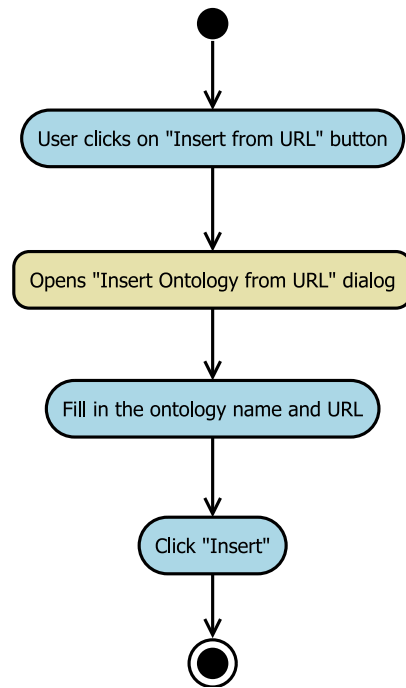


Figure 3.4: Import Ontology

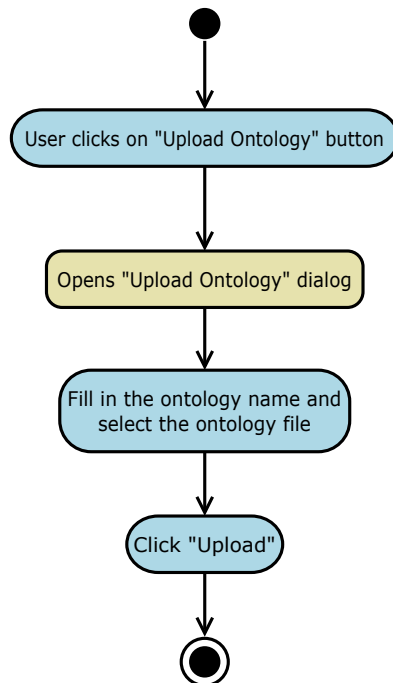


Figure 3.5: Upload Ontology

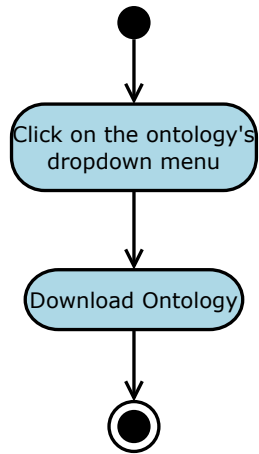


Figure 3.6: Download Ontology

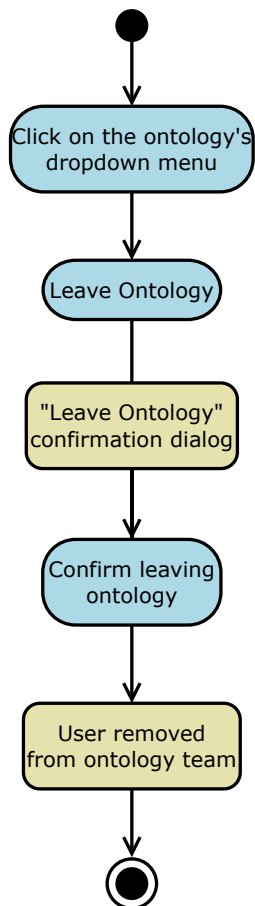


Figure 3.7: Leave Ontology

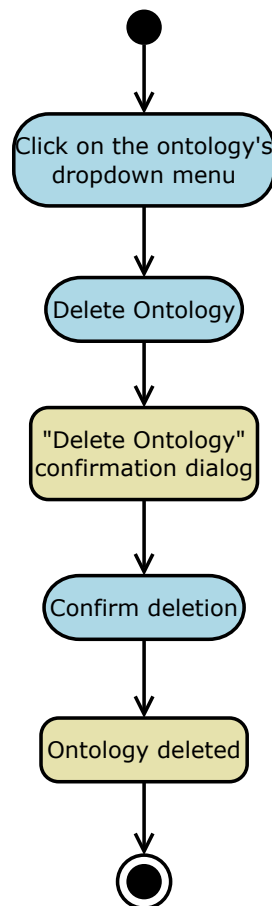


Figure 3.8: Delete Ontology



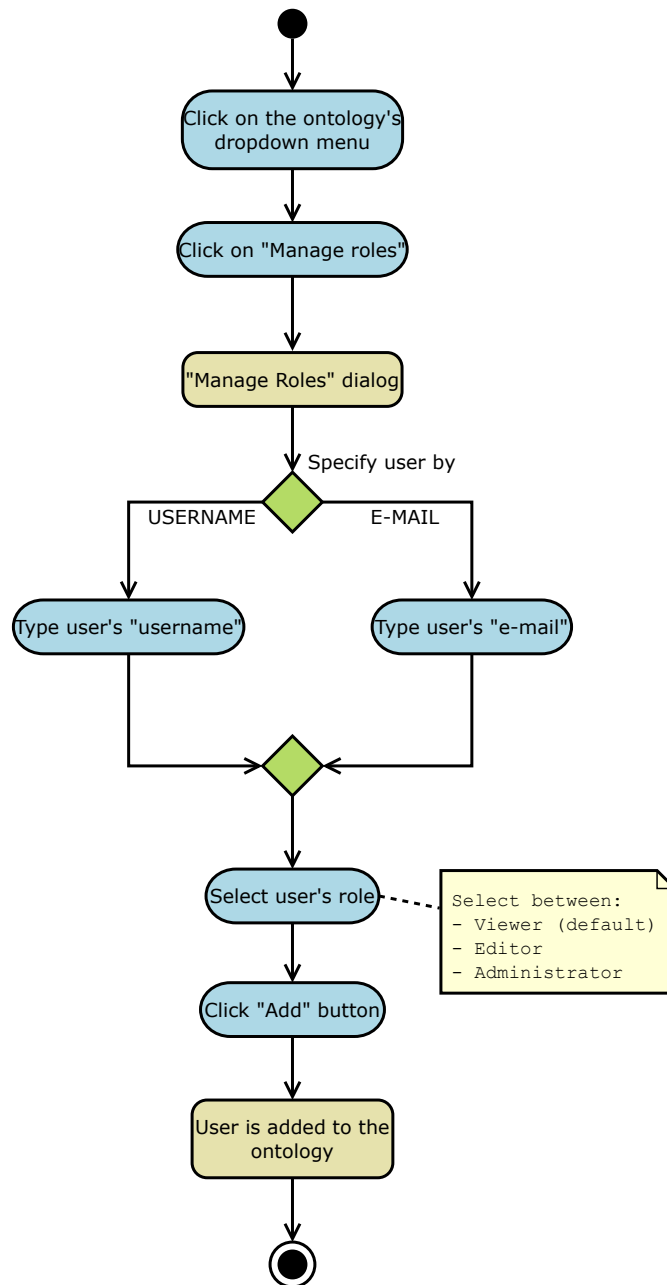


Figure 3.9: The process of granting users with access privileges to the ontology as viewers, editors or administrators

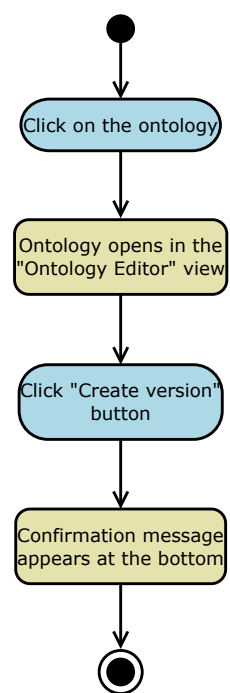


Figure 3.10: Creating new ontology versions. Use case (ix).

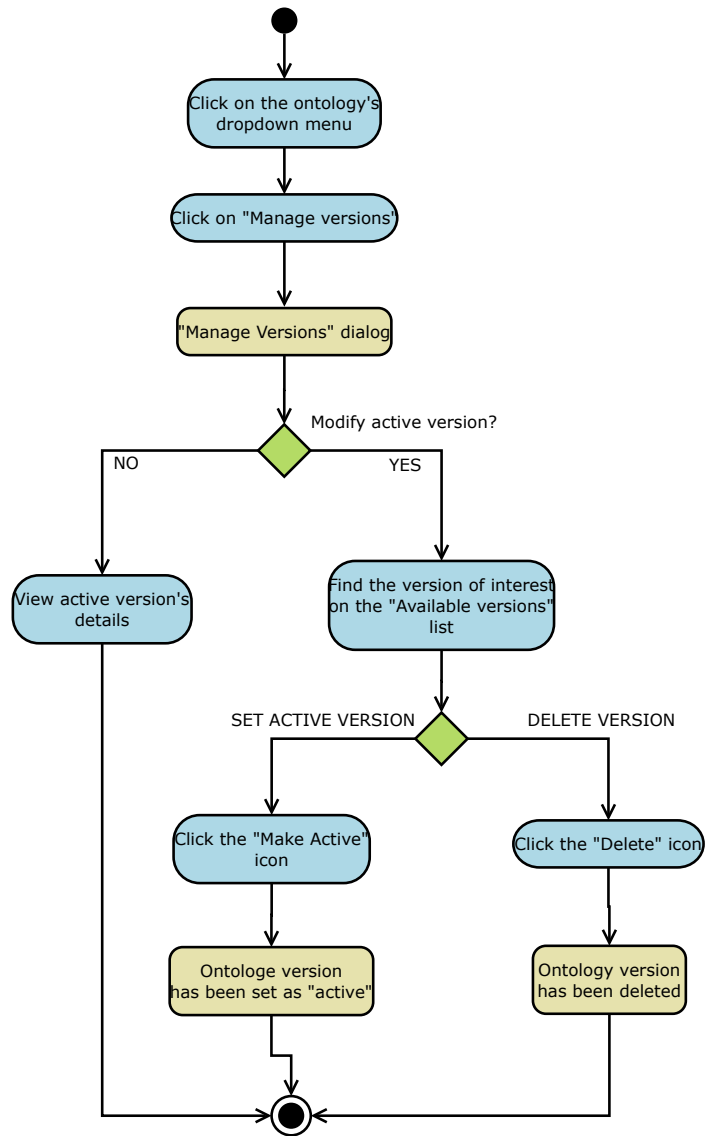


Figure 3.11: Manage ontology versions from the “Manage versions” dialog window. These activities describe the processes of viewing the ontology’s version details, setting a different ontology version as active, or deleting an ontology version.

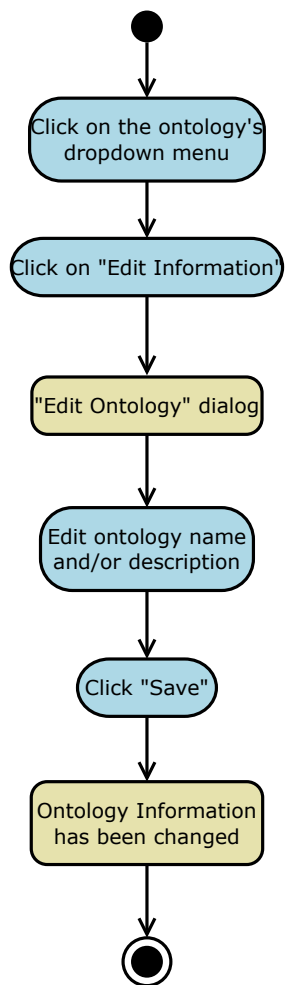


Figure 3.12: Edit an ontology's name or description.

### 3.3.2 Ontology Editing

The “Ontology Editing” activities refer to the use cases described in subsection 3.2.2. In these, the user starts from the ontology editor page and they take the necessary steps to meet their respective goals.

The diagrams used here are UML Swimlane Activity Diagrams. In these diagrams, each actor is represented by a swimlane and the activities that they perform are enclosed into this lane. In our case, we consider the user and CLONE’s component to be the actors.

**Open Ontology Editor:** The user clicks on an ontology listed in the Home page. The *Clone Application* then loads the ontology details (ontology name and location in the *Ontology Repository*) from the database. The *Clone Application* sends the ontology information to the *Ontology Editing Service* (hereafter *OES*), requesting the Ontology Axioms, in JSON format.

In order to serve this request, the *OES* needs to have the latest version of the ontology loaded on its memory (presented as “datastore” in Figure 3.13). It requests the ontology file from the *Repository*, which sends it in RDF/XML format. The *OES*, then, receives the file, stores it to its memory, serializes the ontology axioms to JSON format and returns them to the *Clone Application*. The latter converts the Ontology Axioms to human readable form and displays them to the user.

In the case that another member from the editing-team has already opened the ontology in the editor, the *OES* will have it already loaded on its memory, thus it will serve the request without having to request the ontology file it from the *Repository*.

This activity covers the ontology editing use case (i). and is prerequisite for all other ontology editing activities in this subsection. The UML activity diagram is illustrated in Figure 3.13.

**Edit Ontology:** The ontology has been opened in the Ontology Editor. In this environment, the Ontology entities are grouped in various tabs, based on their type (Classes, Object Properties, Individuals, etc.).

Whenever the user selects an entity tab, the related information is requested from the *Clone Application*. The latter responds with the data and the user is presented with the respective entity list. The user can click on an entity to view any axioms associated with it. This information is again requested by the *Clone Application*, which in turn requests them from the *Ontology Editing Service* (*OES*). The *OES* pulls the requested information from its memory and sends them to the *Clone Application* that displays them to the user.

The user then chooses to edit the selected entity. The changes performed are sent to the *Clone Application*, which forwards them to the

*OES*. The latter applies the changes to the ontology and then performs two simultaneous actions: (a) requests from the *Ontology Repository* to store the changes in the current version's ontology file and (b) responds to the *Clone Application* with the changes performed. The *Clone Application* then responds to the user by displaying the updated entity axioms and, at the same time, notifies any other connected users of the changes that have been performed. This way, CLONE manages to display all user changes in real time and ensures that all active users are displayed the latest version of the ontology.

This activity covers the ontology editing use case (ix). Figure 3.14 illustrates the UML activity diagram.

**Apply Reasoning:** The ontology has been opened in the Ontology Editor. The user clicks on the “Apply Reasoning” button. The *Clone Application* requests from the *Reasoning Service* to apply reasoning on the ontology, providing some ontology details (ontology identifier and a callback URL which will be used for retrieving the inferred ontology, provided that the reasoning process completes successfully).

The *Reasoning Service* retrieves the ontology from the *Ontology Editing Service* and applies reasoning. In case the ontology is found inconsistent, the *Clone Application* is notified and it, in turn, notifies the user. In the case that the ontology is consistent, the *Reasoning Service* simultaneously performs two actions: (a) Notifies the *Clone Application* of the result and (b) sends the inferred ontology to the *OES* in order to store it and make it accessible to the *Clone Application*.

The user may select to view the inferred ontology. The *Clone Application* requests the ontology's JSON representation from the *OES* and when it receives it, it displays the ontology to the user (in a separate window).

This activity covers the ontology editing use cases (iii), (iv) and (v). Figure 3.15 illustrates the UML activity diagram.

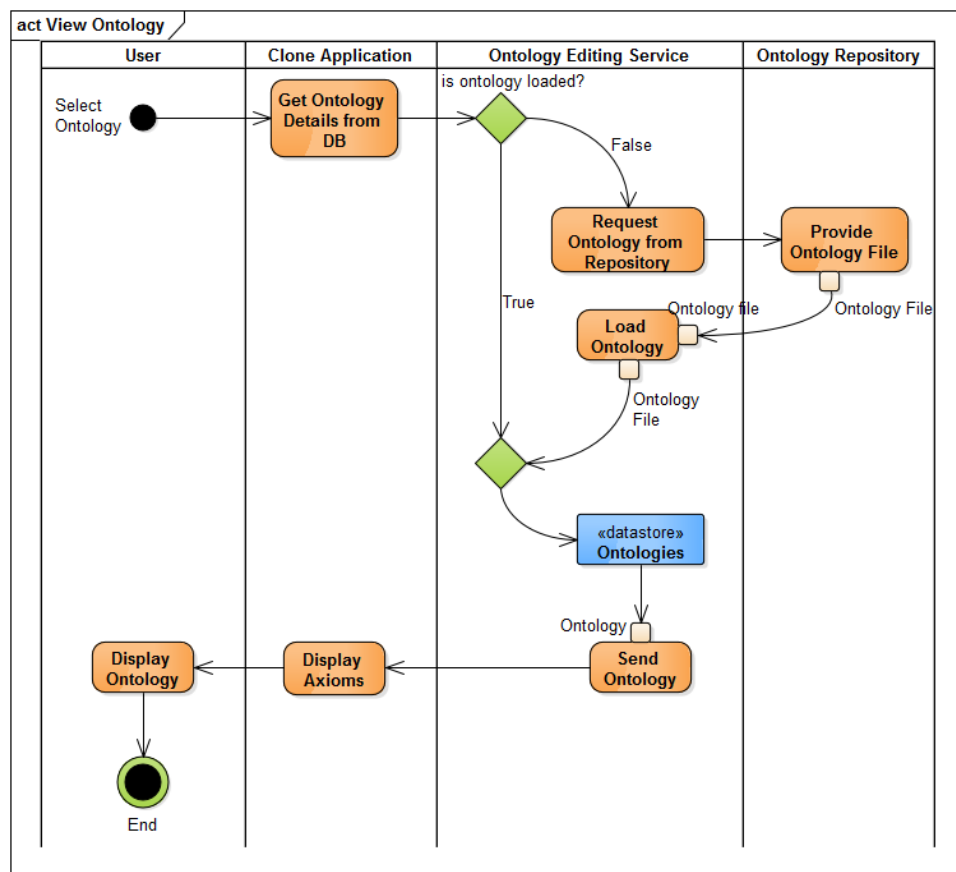


Figure 3.13: Activity diagram describing the process of opening an ontology for editing. This step is prerequisite for all other ontology editing activities in this subsection.

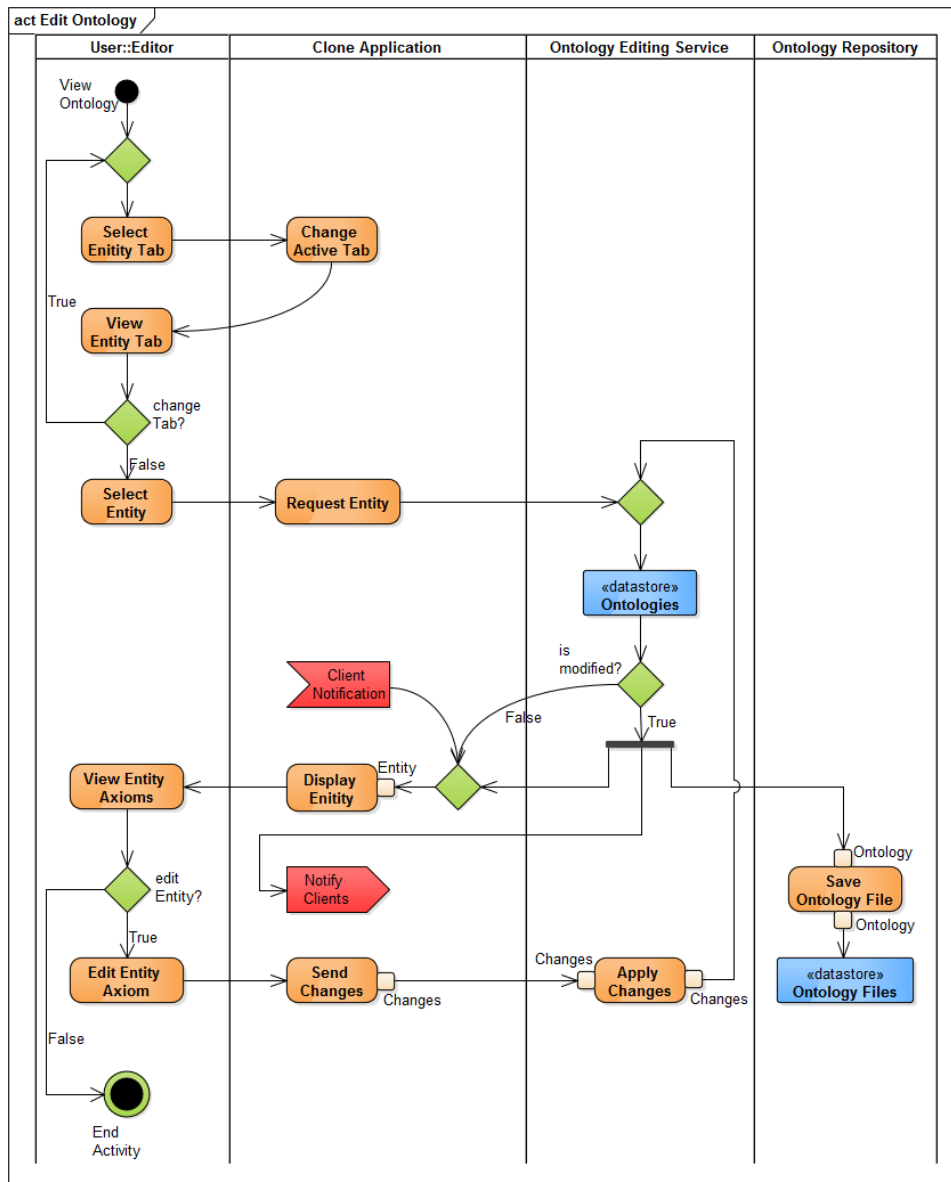


Figure 3.14: Activity diagram describing the process of editing entities of an ontology.



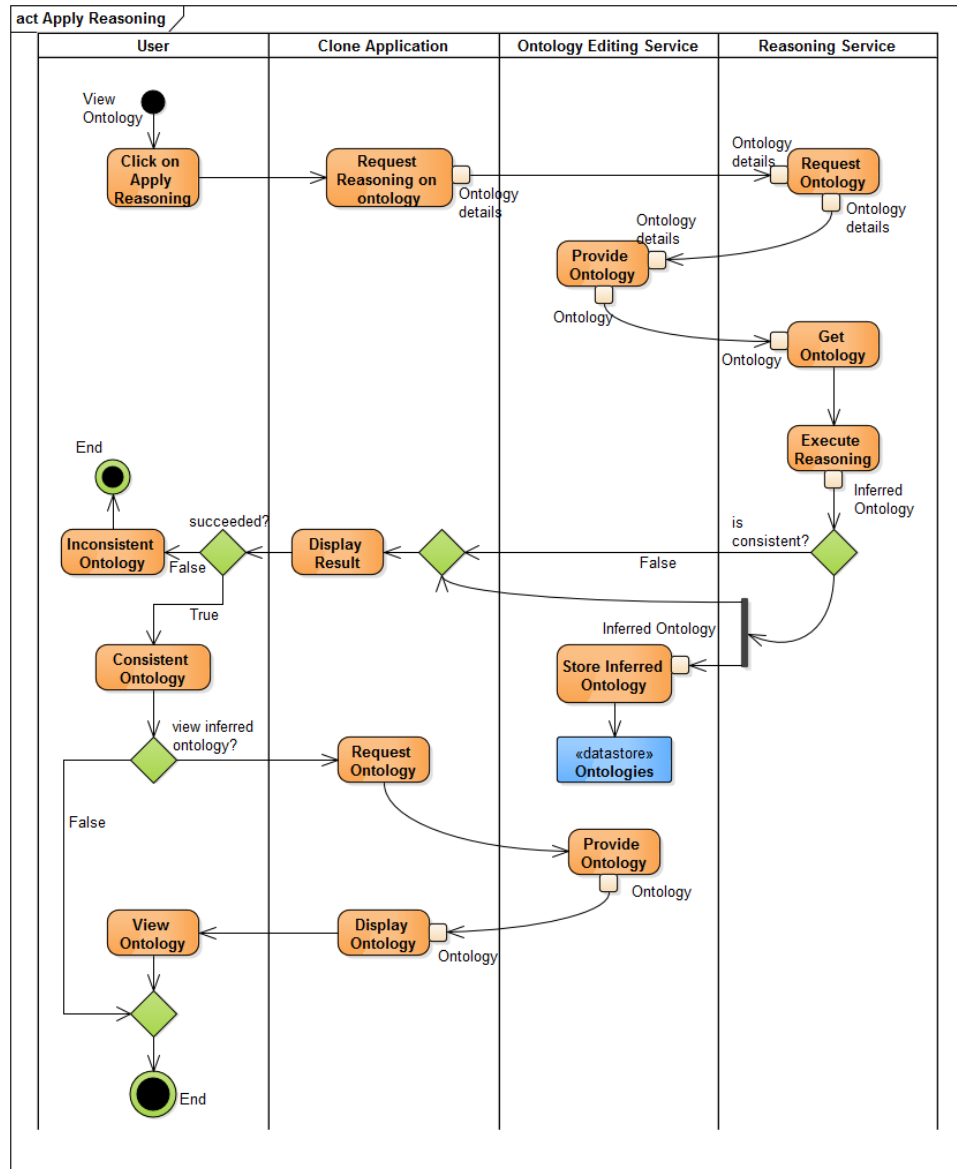


Figure 3.15: Applying reasoning on an ontology.

## Chapter 4

# Architecture

In this chapter we identify the logical layers that group the system's operations and we provide an in-depth analysis of the software components that constitute it.

### 4.1 Challenges and Decisions

CLONE has been implemented taking into consideration the following requirements:

- (i) **High demand on computational resources.** CLONE has to cope with multiple users editing large ontology files simultaneously. Several editing teams may use the application at the same time, updating their ontologies, performing reasoning and participating in conversations. These tasks, especially when performed by several users simultaneously, require large amounts of computational resources, something that the system has to deal with.
- (ii) **High Availability.** Specific features supported by CLONE can be very time-consuming. For instance, reasoning over an ontology is a task that may take several minutes to complete. CLONE has to fulfill such operations, without compromising the service quality for other users that are using the system at the same time.
- (iii) **Extensibility.** Though CLONE fully supports OWL 2, there might be a need to further extend its functionality either to enhance the user experience or to add more features. To that end, the system has to be modular and easily extensible.

To address these challenges, CLONE has been designed as a component-based, service-oriented, distributed system that enables:

- (i) **Distribution of Workload.** The system comprises of different independent service components, communicating with each other using

REST APIs. This design allows the distribution of the workload to different virtual machines. It also helps to achieve high availability, since the single point of failure is eliminated. In the case of a component failure (e.g. the ontology conversation service becomes unavailable) users will still be able to use other features of the application.

- (ii) **Asynchronous execution of long processes.** The components responsible for executing long processes (such as storing ontology files), have been implemented as asynchronous Web Services that are executed silently in the back-end, allowing users continue working on the ontology uninterrupted.
- (iii) **Extensibility.** Since the components operate independently, it is very easy to modify them, replace them or add totally new components, without compromising the operation of the entire system.

## 4.2 Architectural Layers

CLONE adopts a 3-layer architecture design model. The first layer (Presentation Layer) handles the user's interaction with the system. The second layer (Application Layer) implements the business logic of the system and manages the communication between the various service components. Finally the third layer (Data Access Layer) is responsible for persisting information, be storing it relational or document oriented databases or in the file system. These layers are illustrated in Figure 4.1 and are discussed in the following subsections.

### 4.2.1 Presentation Layer

The Presentation Layer is responsible for interacting with the User. It implements functionality for:

- Providing the appropriate methods (User Interface) that enable the user to interact with the system.
- Transferring user commands to the Application Layer in order to be executed by the back-end.
- Presenting the information provided from the back-end to a human-readable form.

The Presentation Layer implements also the Web interface for users interacting with the system. It communicates with the Application Layer over REST APIs. The data exchanged between the communicating layers is in JSON format. In specific use cases (such as Editing an Ontology) the exchanged data need to be translated from JSON to HTML or reverse. This transformation is handled by the *Transformation Component*.

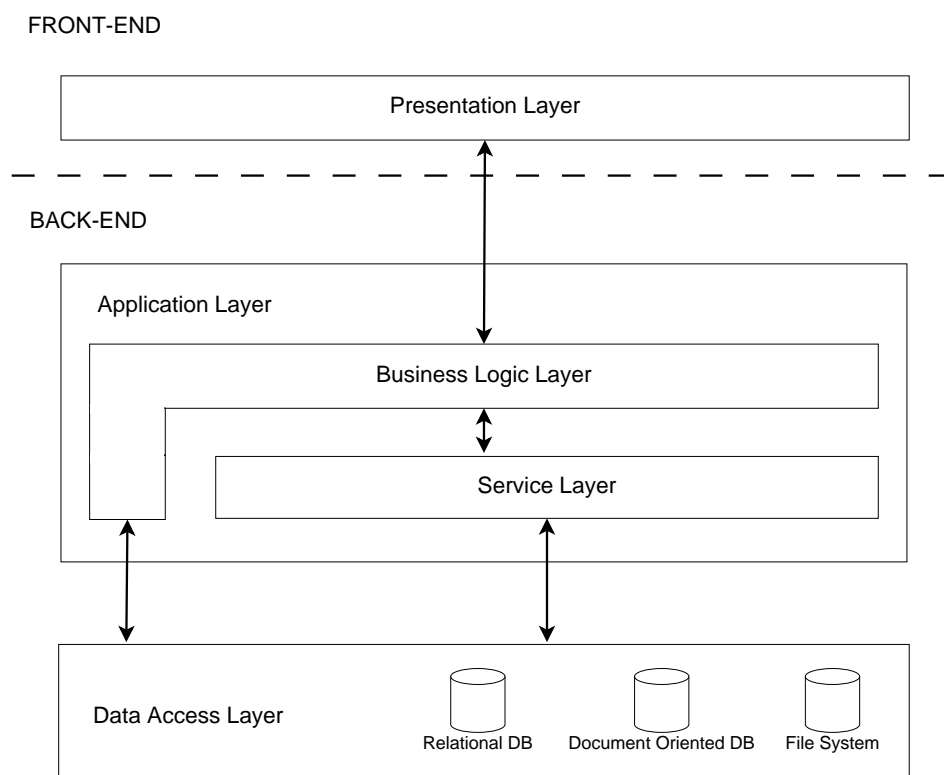


Figure 4.1: Architectural Layers

### 4.2.2 Application Layer

This layer operates in the back-end and is responsible for processing information. It can be further divided into two sub-layers:

- (i) The Business Logic Layer that is responsible for executing the business logic of the application. It serves as the orchestrator of the entire application, as it handles the communication among the other layers.
- (ii) The Service Layer that includes a pool of services that are utilized by the previous sub-layer. Each of the individual services are discussed in the following subsection.

### 4.2.3 Data Access Layer

This layer also operates in the back-end. Its responsibility is to store and retrieve information that is sent either by the Application Layer or the Service Layer.

## 4.3 Software Components

Each of the layers comprises of a logical group of software components that implement specific functionality. Figure 4.2 illustrates the components that constitute the layers and their inter-connections. In the following subsections we discuss each software component and the functionality that it implements.

### 4.3.1 Web UI Component

The set of the visual elements that are used for displaying information to the user and accept user input. These include the Web pages of the Application, as well as the scripts and modules that are responsible for presenting of information.

### 4.3.2 Transformation Component

The data that is exchanged between the Presentation Layer and the Application Layer is in JSON format. The module responsible for transforming the data into JSON is the Transformation Component. More specifically, it performs the following transformations:

- (i) Data received from the back-end into an appropriate format to be presented to the user. In particular, it receives data in JSON format from the Application Layer and transforms it to HTML to be presented to the user.

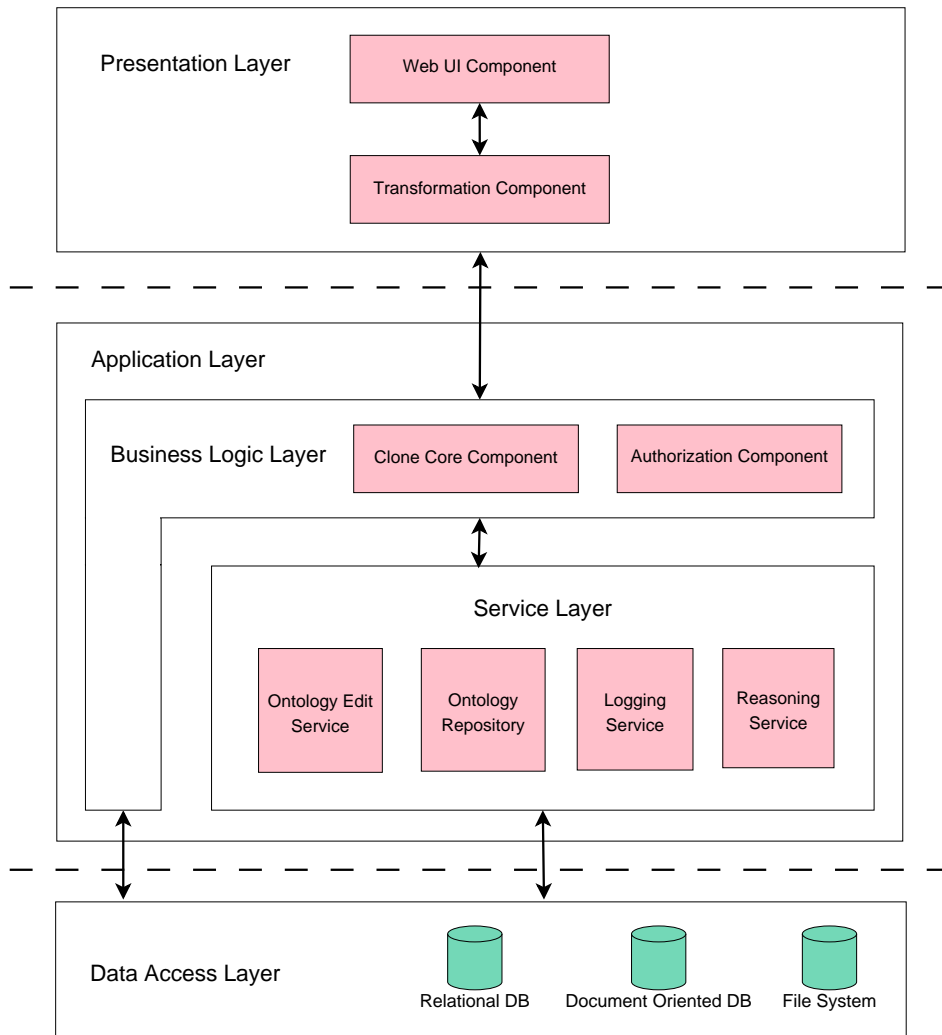


Figure 4.2: Architectural Layers with Components.

- (ii) User actions and commands into a format that can be processed and executed by the back-end. More specifically, it transforms user commands into JSON objects before they are sent to the Application Layer.

### 4.3.3 Authorization Component

This component is responsible for user authentication and authorization. It performs user identification based on the provided credentials<sup>1</sup> and grants the appropriate permissions to access specific resources or perform specific actions. Assigning these functionalities to a separate component, allows for easily integrating different authorization methods (i.e., OAuth2) or even use third party authorization services as alternatives (e.g., Facebook, Google Account etc.).

### 4.3.4 Clone Core Component

Clone Core Component is responsible for executing the business logic of the application. It works as the orchestrator of the system, as it receives requests by the user, divides it into simpler tasks and distributes them to the other service-components. Afterwards, it combines the results of each separate task to compose the response that will be sent to the user. More specifically, the responsibilities of this component are:

- (i) Provide the interface through which the front-end communicates with the back-end.
- (ii) Receive the user's requests, and partially process the information.
- (iii) Forward received information to the appropriate services that will process it.
- (iv) Respond to the front-end with the outcome of the information processing.

The architecture of Clone Core Component is illustrated in Figure 4.3. It implements the following software modules:

- (i) **User Authentication** whose role is to identify the user. This module is responsible for logging the user in the system and initialize the user's session.
- (ii) **Authorization Component** whose purpose is to check the user's permissions before the execution of any action and either allow the execution, provided that the user has the appropriate permissions, or block it otherwise. In the latter case, the component also generates

---

<sup>1</sup>In the current implementation, the component performs username-password authentication.

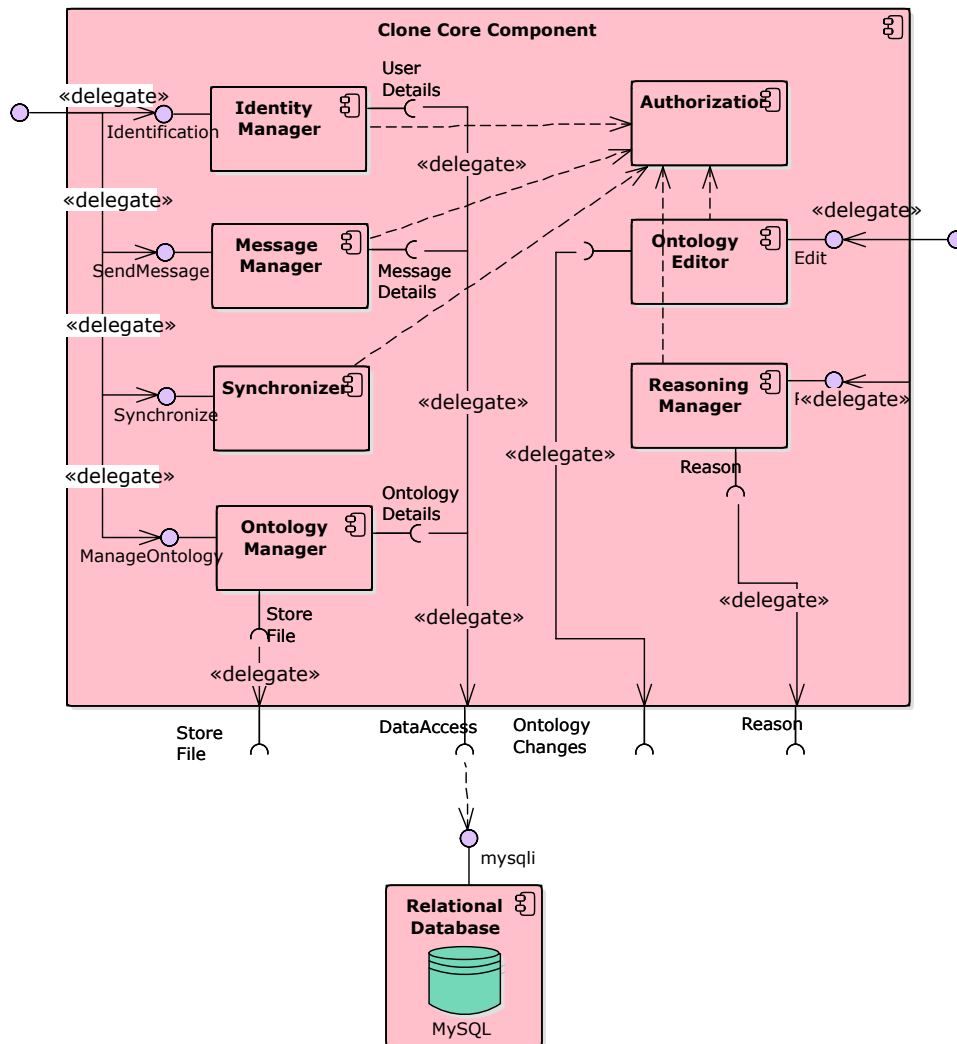


Figure 4.3: Architecture of the Clone Core Component.



an error message. This component operates as a middleware and is invoked before serving any requests. It cannot be directly invoked by the user but only by other components that require authorization.

- (iii) **Ontology Manager** that handles all user's interactions with Ontology projects (not Ontology contents). This module provides the functionality to create, edit or delete Ontology projects. Furthermore, it is used for granting or revoking user permissions on ontology projects. The component interacts with CLONE's SQL database (described in Appendix A.1), to persist changes on the ontology project information, as well as with the *Ontology Repository* for creating/deleting the associated ontology files.
- (iv) **Ontology Editor Manager** that is responsible for communicating with the Ontology Editing Service (OES) and forward any changes the user made to the ontology contents. The component implements the functionality that enables connecting to the OES over its REST API, to perform actions that the user has requested on a specific ontology.
- (v) **Reasoning Manager** that is responsible for carrying forward the user's request for reasoning over the ontology to the Reasoning Service.
- (vi) **Message Manager** that is responsible for delivering the messages that are exchanged in the Ontology Editor's chat. It utilizes the Server Sent Events (SSE)<sup>2</sup> technology enabling real-time message transmission without requiring from the client to sent multiple requests.
- (vii) **Synchronizer** that is responsible for synchronizing the clients that are simultaneously editing the same ontology file. Each change that a user makes in the ontology, is transmitted through this component, to all the clients that are viewing the specific ontology's contents. The Synchronizer also utilizes the SSE technology.

As its name suggests, Clone Core Component is the core of the entire system, since it is responsible for coordinating the software components and also executing the business logic. It can easily be inferred that this component can only exist once in each deployment of CLONE.

#### 4.3.5 Ontology Editing Service

Ontology Editing Service is the component responsible for editing ontologies. It has been designed as a RESTful Web Service, independently from the other components. The component has been written in Java Programming

---

<sup>2</sup><http://www.w3.org/TR/eventsource/>

Language<sup>3</sup>, utilizing the JAX-RS API<sup>4</sup> for implementing its REST API and, internally, the OWL API for manipulating the ontologies. Its architecture is illustrated in Figure 4.4. It comprises of the following software modules:

- (i) **Ontology Loader** is responsible for retrieving ontology files from specified locations and for loading the ontologies to its memory, making them available for editing. The ontology file has to be accessible on the Web. Once an ontology has been loaded, its entities can be accessed for manipulation through the component’s REST API.
- (ii) **Ontology Editor** performs read and write operations on the resources and communicates with the Ontology Saver in order to store the changes in the ontology file.
- (iii) **Ontology Saver** is the module responsible for writing the ontology to a file and then send the file in order to be stored. The ontology file is sent over the HTTP, to the location that the ontology has been retrieved from, using the PUT method. In the scope of our implementation the location where the ontologies are retrieved from or stored to, is the URL of the Ontology Repository component.
- (iv) **Inactive Ontology Collector** is used for unloading from the system’s memory any “unneeded” ontologies. It operates as a background process that is invoked periodically (every 15 minutes) and checks, for each loaded ontology, the last time that it has been accessed. The ontologies that have not been accessed for more than an hour are removed from the systems memory. This module is essential to the system’s sustainability, as it frees computational resources and makes them available for reuse.

The Ontology Editing Service handles the changes that are taking place in the ontologies that are edited by CLONE. Since the ontologies are loaded into the components memory, in the case that multiple ontologies are edited simultaneously, the memory resources that are occupied may be massive (depending on the size of each ontology). In that end, Ontology Editing Service has been designed to operate independently, allowing it to be replicated and enabling load-balancing among the replicas.

#### 4.3.6 Ontology Repository

The Ontology Repository is a Web Service responsible for managing ontology files. It provides a REST API that enables the creation and deletion of ontology files and ontology versions. The ontology files are stored into the server’s file-system. Each ontology file is stored along with a metadata object

---

<sup>3</sup><https://www.java.com/en/>

<sup>4</sup><https://github.com/jax-rs>

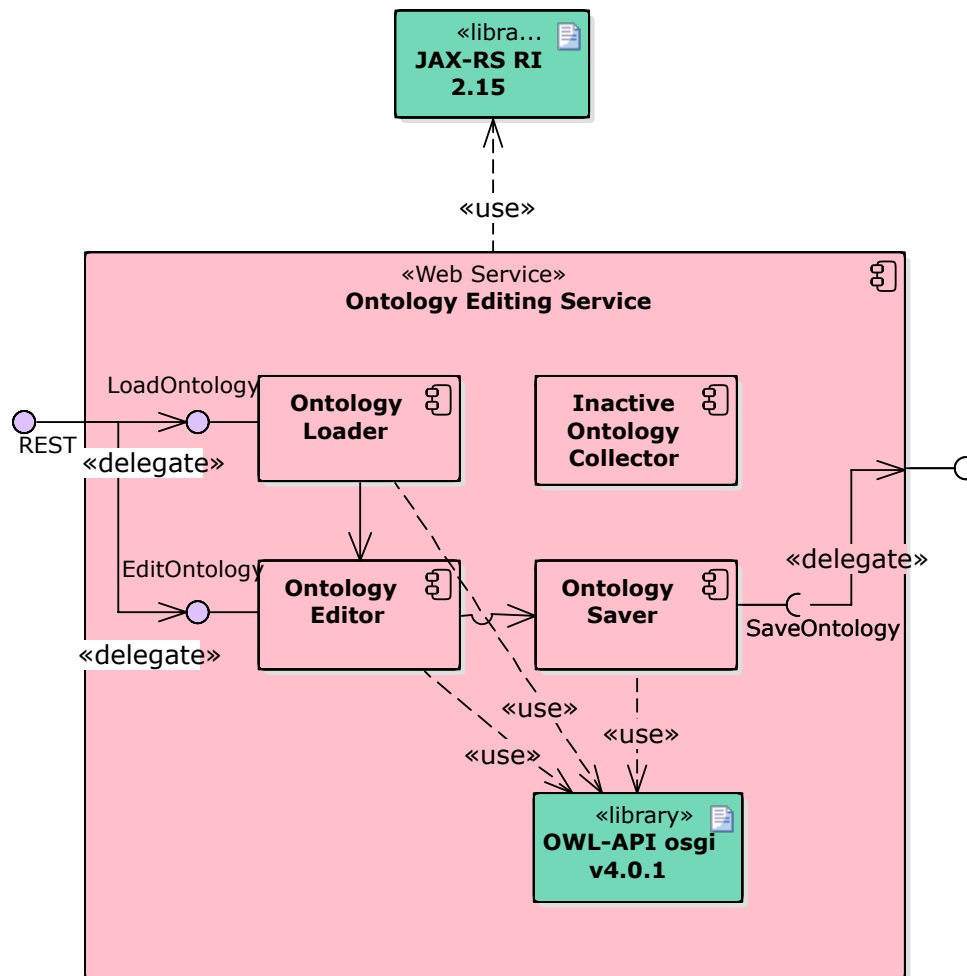


Figure 4.4: Architecture of the Ontology Editing Service.

Listing 4.1: Ontology metadata stored by the Ontology Repository

```
1 {  
2   "databaseId": "15e67a1090de6d",  
3   "fileName": "Family",  
4   "resourceUri": "http://147.27.60.63:8080/OntologyRepository  
    ↪ /webresources/ontologies/15e67a1090de6d",  
5   "creator": "apreventis",  
6   "creationTime": "2020-03-15_14:52:46",  
7   "activeVersion": false  
8 }
```

that contains information about the ontology. The metadata objects are stored in JSON format (displayed in Listing 4.1) into a MongoDB database. They store the following information:

- A unique ontology project ID, named “databaseId” that is used for associating the metadata objects with an ontology project. Many metadata objects may be associated with the same ontology project, but each metadata object corresponds to exactly one ontology file in the repository. This is because, an ontology project is associated with as many ontology files as the versions that have been saved by its editing team.
- The file name of the ontology. This name is used when the user requests to download the ontology to their local machine.
- The resource location on the Web.
- The username of the creator of the ontology file.
- The date and time that the file was created.
- An indicator that shows whether the ontology file is the one that is currently used as the active version in the ontology project. Only one active version may exist in a project.

Figure 4.5 illustrates the components of the Ontology Repository. These are:

- (i) **Ontology Manager** provides a REST API for accessing ontology files. It handles the requests for the specified resources (i.e., ontology files) and invokes the Metadata Manager and the File System Manager.
- (ii) **Metadata Manager** is responsible for interacting with the component’s NoSQL database, in order to manage the metadata objects. These include creating new metadata objects whenever new ontology files are added to the repository, updating existing metadata objects

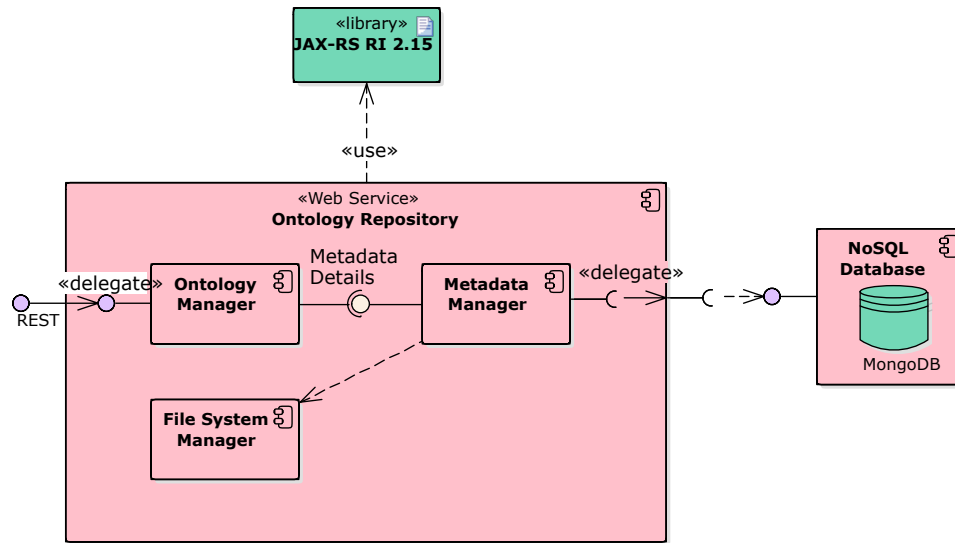


Figure 4.5: Architecture of the Ontology Repository Service.

when the active version of an ontology is changed or deleting metadata objects when their associated ontology file is removed.

- (iii) **File System Manager** handles the interactions with the ontology files. It is responsible for organizing the ontology files in the file system’s directories, and also for creating, retrieving or deleting ontology files.

The Ontology Repository has been written in Java, using the JAX-RS library for providing the REST features.

#### 4.3.7 Reasoning Service

The Reasoning Service is a Web Service that provides an API for applying Reasoning on ontology files. When invoked, it requires two URLs, the first (named “ontologyUri”) specifies the location of the ontology to be reasoned about and the second (named “callbackUrl”) specifies the location where the reasoned ontology will be sent, after the reasoning process has completed. The service pulls the ontology from the Web location specified and loads it into its memory, where the reasoning takes place. After the reasoning is completed and in the case that the ontology was found consistent, the service will send the inferred ontology (using a POST request) to the second URL that was provided. Finally the service will respond to the requester, informing on the reasoning results and providing the Web location of the inferred ontology. Listings 4.2 and 4.3 display examples of request and response payloads, to the Reasoning Service, respectively. Both the payloads are in JSON format.

Listing 4.2: Request payload for invoking the Reasoning Service

```

1 {
2   "callbackUrl": "http://147.27.60.63:8080/
   ↳ OntologyEditingService/rest/ontologies",
3   "ontologyUri": "http://147.27.60.63:8080/
   ↳ OntologyEditingService/rest/ontologies/10"
4 }

```

Listing 4.3: Response of reasoning over a consistent ontology

```

1 {
2   "inferredOntologyUri": "http://147.27.60.63:8080/
   ↳ OntologyEditingService/rest/ontologies/11",
3   "isConsistent": true
4 }

```

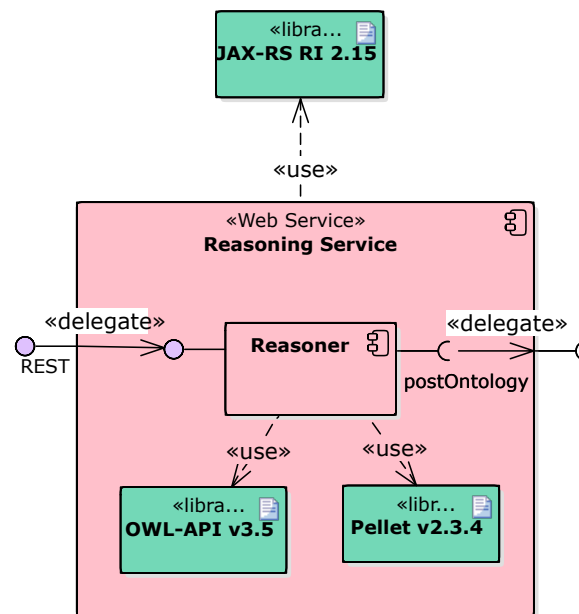


Figure 4.6: Architecture of the Reasoning Service.

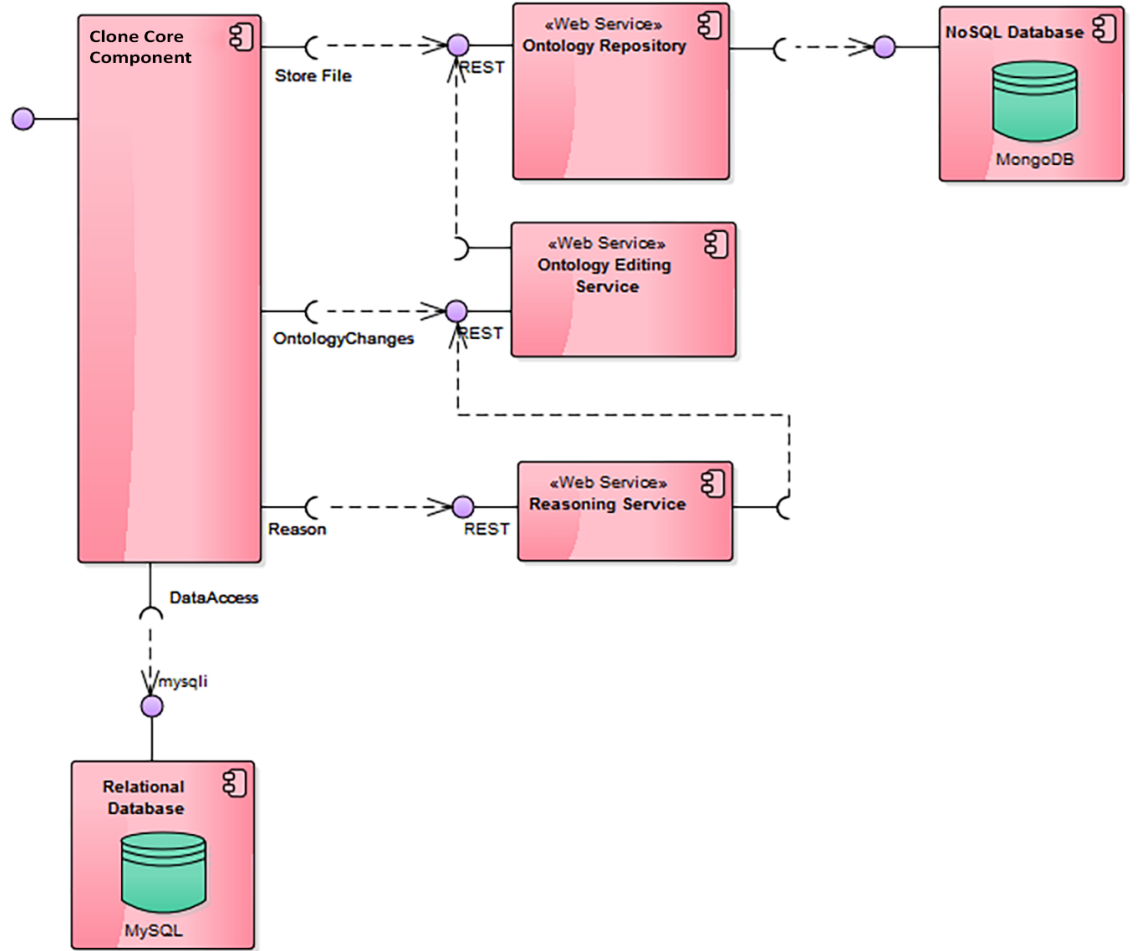


Figure 4.7: Overall Architecture

## 4.4 Overall Architecture

In the previous section, we have described CLONE's software components and the functionalities that each of them implements. Figure 4.7 illustrates how these components are connected and interoperate in order to provide CLONE's functionality.

The user interacts with the *Clone Core Component*, which is responsible for receiving each action and breaking it into smaller simpler tasks. These tasks are then distributed to the rest of the components, over REST API calls. When a component completes its task, it responds to the *Clone Core Component* with the results. Finally, when all components have responded, the *Clone Core Component* will combine the individual responses into the final response that will be presented to the user.

## Chapter 5

# Implementation

### 5.1 Deployment

In the previous chapter we introduced the software components that constitute CLONE. In this section we describe the deployment targets for each of these components, that in the UML notation are called *Nodes*<sup>1</sup>.

In CLONE's implementation each node is a Web Server, that allows a component to work independently as a Web Service and inter-operate with other components, by making its REST API available on the Web. Figure 5.1 illustrates the nodes where the software components have been deployed. CLONE consists of the following nodes:

- (i) **Clone Application Server** is the node that receives all requests directly from the user and communicates with all other nodes utilizing the services they provide. This node includes an Apache HTTP Server where the *Clone Core Component* is executed and a MySQL Database Server that hosts CLONE's relational database.
- (ii) **Ontology Repository Server** is the node responsible for storing ontology files and versions. This server includes an Apache Tomcat Web Server, that executes the *Ontology Repository* component and a MongoDB Server that stores the ontology file metadata.
- (iii) **Ontology Manipulation Server** includes an Apache Tomcat Web Server, that executes the *Ontology Editing Service* component.
- (iv) **Ontology Reasoning Server** also includes an Apache Tomcat Web Server, that executes the *Reasoning Service* component.

---

<sup>1</sup>A Node is a deployment target which represents computational resource upon which artifacts may be deployed for execution.



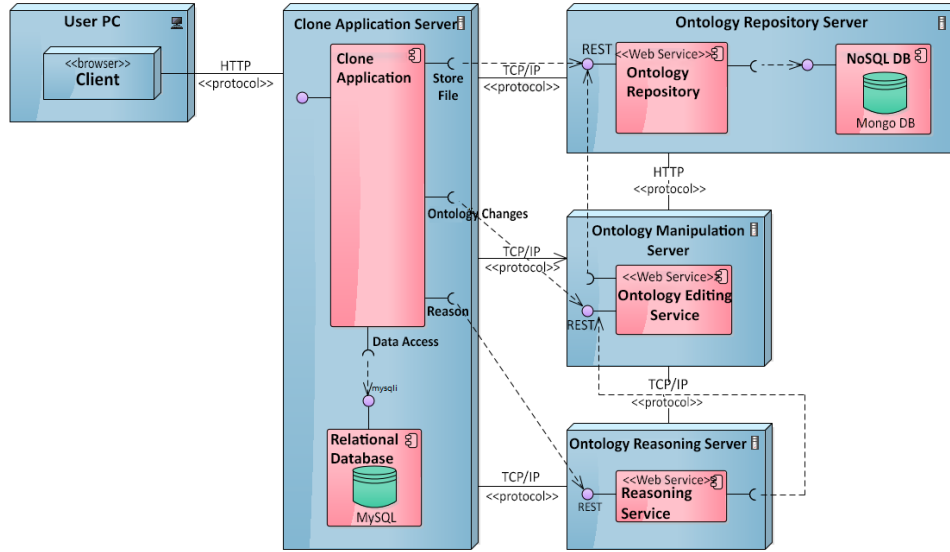


Figure 5.1: The nodes where CLONE's artifacts have been deployed.

## 5.2 User Interface

CLONE's User Interface (UI) has been designed given emphasis on both usability and functionality. It enables users to easily create, manage and manipulate Ontologies using OWL's expressiveness to its full extent.

The layout and color conventions have been inspired by Protégé OWL Editor, one of the most popular ontology editors [22]. This way, users can be familiar with the tool, even if they are using it for the first time.

In this section we describe the pages of the application, along with the functionality that is provided by each one.

### 5.2.1 Sign-in

When a user first visits CLONE, the landing page is displayed (Figure 5.2) that has two forms: one for logging into the system, provided that the user has already created an account, and another for registering to the system and creating new account. All fields in each of these forms are required. When the user fills either form, he/she will enter the application and is redirected to the home page.

### 5.2.2 Home page

The home page (Figure 5.3) is the first page that the user views after logging into the system. The page consists of two basic visual components (a) the header, which is (the dark area) placed on the top of the page and (b) the content, which is the white area in the center of the page. The header is

Figure 5.2: The user fills-in either the left form to log-in or the right in order to create an account and acquire credentials.

Figure 5.3: The user's home page.

fixed and it is the same in all the application pages and from there the user can navigate in the application, or log-out from it. The content is the visual component where the dynamic information of each page is displayed and it varies from page to page.

On the top of the page's content there are the "New ontology" buttons. CLONE provides three ways to create an ontology: create an ontology from scratch, import an ontology from a web location or upload an ontology from the local file system. The different options to create ontologies are described below.

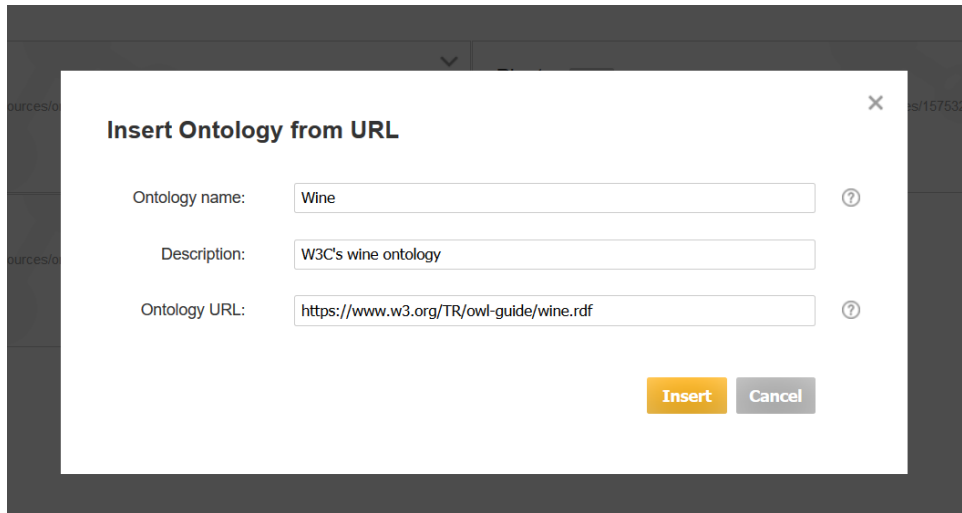
- (i) **Create ontology:** When the user clicks on the "Create Ontology"

Figure 5.4: The modal for creating a new ontology.

button, a modal will appear containing a form, as shown in Figure 5.4. The form contains three fields, (a) the ontology name, a name that will help the user and their team to identify the ontology, (b) the ontology IRI (an ontology IRI will be pre-filled by the system, but the user may change it as they will) and (c) a description of the domain that the ontology represents.

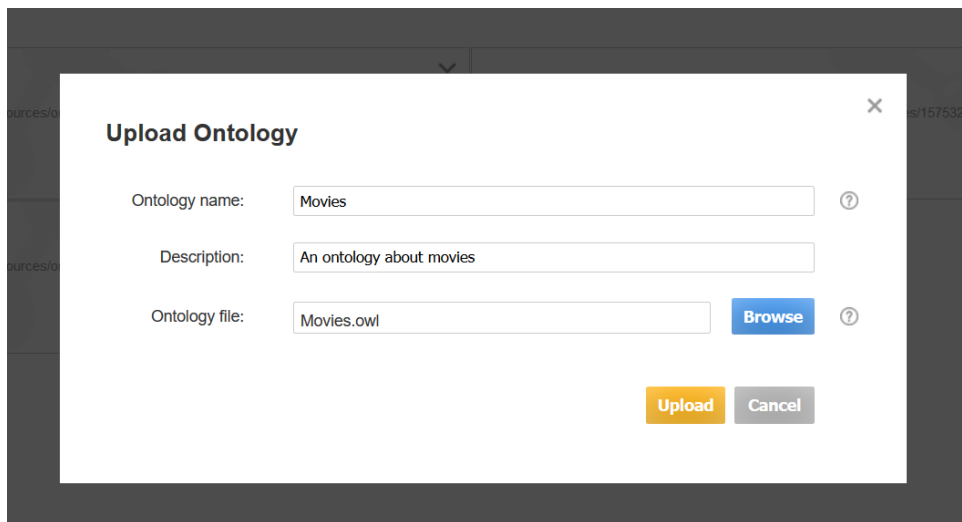
- (ii) **Insert from URL:** This option allows the user to import an already existing ontology from a web location. When the user clicks the “Insert from URL” button a modal will appear (Figure 5.5), where the user has to specify the ontology name, an optional description of the ontology and the URL of the ontology. After clicking “Insert” the system will create a copy of the ontology from the specified location and store it into the Ontology Repository, making it available for manipulation to the user and the development team.
- (iii) **Upload ontology:** Finally, by clicking on the “Upload ontology” button, a modal will appear where the user has to specify the ontology name, an optional description for the ontology and select the ontology file from the local file system, as displayed in Figure 5.6. Then, by clicking “Upload” the specified ontology file will be copied to the Ontology Repository and the ontology will be available for editing.

After creating an ontology using any method, the respective modal will automatically close and the new ontology will be added in the user’s list. The creator of the ontology will be assigned the administrator role, which cannot be changed.



The modal is titled "Insert Ontology from URL" and features a close button (X) in the top right corner. It contains three input fields: "Ontology name:" with the value "Wine", "Description:" with the value "W3C's wine ontology", and "Ontology URL:" with the value "https://www.w3.org/TR/owl-guide/wine.rdf". Each input field has a help icon (question mark) to its right. At the bottom right, there are two buttons: "Insert" (orange) and "Cancel" (gray).

Figure 5.5: The modal for importing an ontology from a web location.



The modal is titled "Upload Ontology" and features a close button (X) in the top right corner. It contains three input fields: "Ontology name:" with the value "Movies", "Description:" with the value "An ontology about movies", and "Ontology file:" with the value "Movies.owl". The "Ontology file:" field has a "Browse" button (blue) to its right. At the bottom right, there are two buttons: "Upload" (orange) and "Cancel" (gray).

Figure 5.6: The modal for uploading an ontology from the local file system.

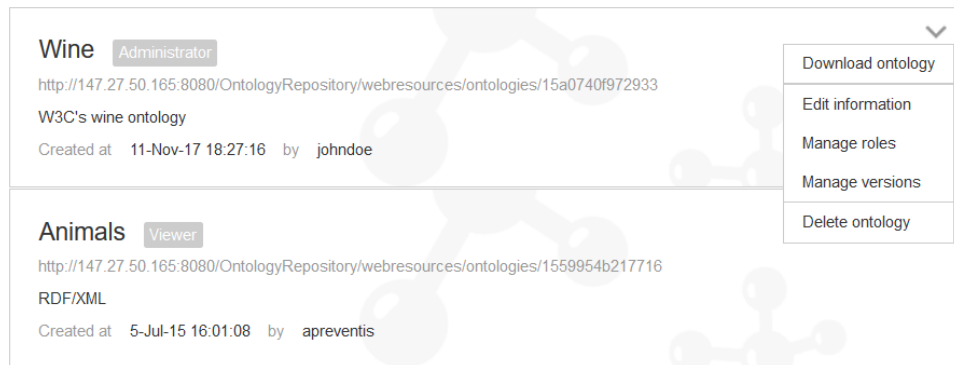


Figure 5.7: The available actions that an administrator can perform on an ontology.

In the home page users can browse the ontologies that they are allowed to view or edit. For each ontology in the list, some basic information is presented such as the ontology's name, its location in the Ontology Repository, a description, its creation date and the user's role in the development team. Users can also perform specific actions on these ontologies, depending on their role in the development team, as they are described in Subsection 3.2.1. The available actions can be selected using the ontology menu, that appears by clicking the expand button on the top right corner of the ontology. Figure 5.7 illustrates the actions that are available on an ontology that the user administrates.

- (i) **Download Ontology:** By selecting this action, the user can download the the ontology file locally. The downloaded ontology file will be saved in the RDF/XML serialization format.
- (ii) **Edit Information:** This option allows the user to edit the basic information of the ontology, such as the ontology name or its description, in a new modal, as displayed in Figure 5.8.
- (iii) **Manage Roles:** This option enables the user to manage the users that can access an ontology and their permissions. When the user selects to manage the ontology's roles, a modal will appear (Figure 5.9) allowing them to (a) add new users to the development team and assign them roles or (b) remove existing users from the development team and revoke their permissions.

A new user can be added to the ontology team by entering his/her username or e-mail in the user input and selecting the role that he/she is going to be associated with, from the roles drop-down. The available



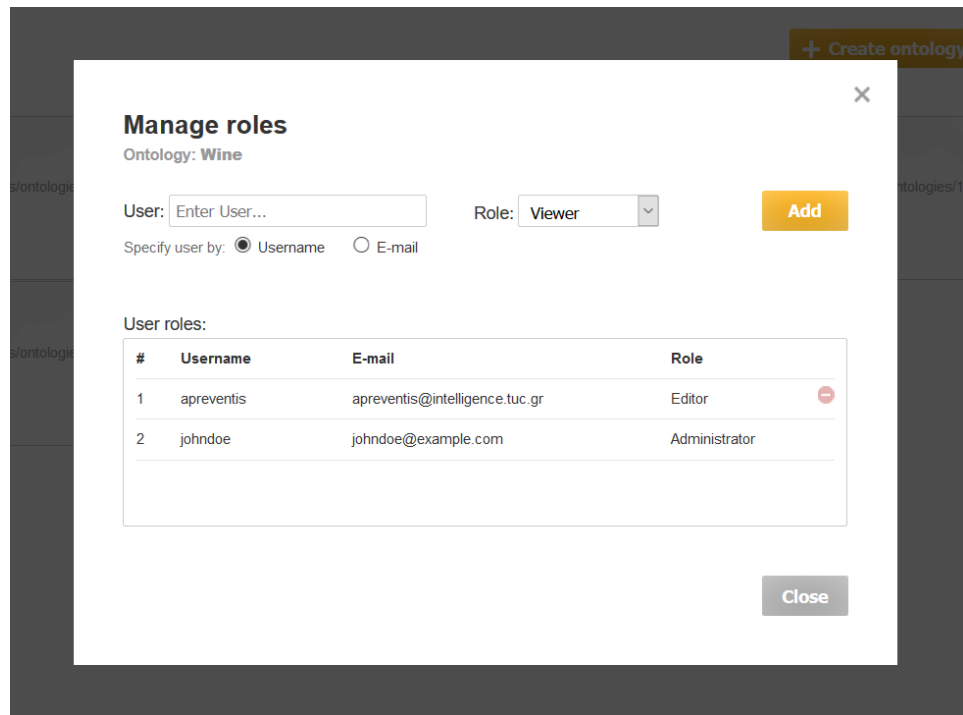


Figure 5.9: Manage user roles.

- (v) **Delete Ontology:** This option allows the user to completely remove the ontology and anything associated with it, including ontology versions and conversations. The ontology will also be removed from the ontology repository.

### 5.2.3 Ontology Editor

In order to open an ontology in the editor, the user has to click on it from the home page, as displayed in Figure 5.3. After this, the user is directed to the ontology editor page, where he/she can view the the ontology data and, provided that they have the appropriate permissions, edit it.

The ontology editor page can be divided in the following visual components:

- (i) **The header**, which is placed on the top of the page and under the application header. The header displays the name of the ontology that is currently open on the editor, and on its right side, the available management actions that the user can perform. These include applying reasoning on the ontology, creating a new version with the current state of the ontology or downloading an ontology file (.owl) that contains the ontology.

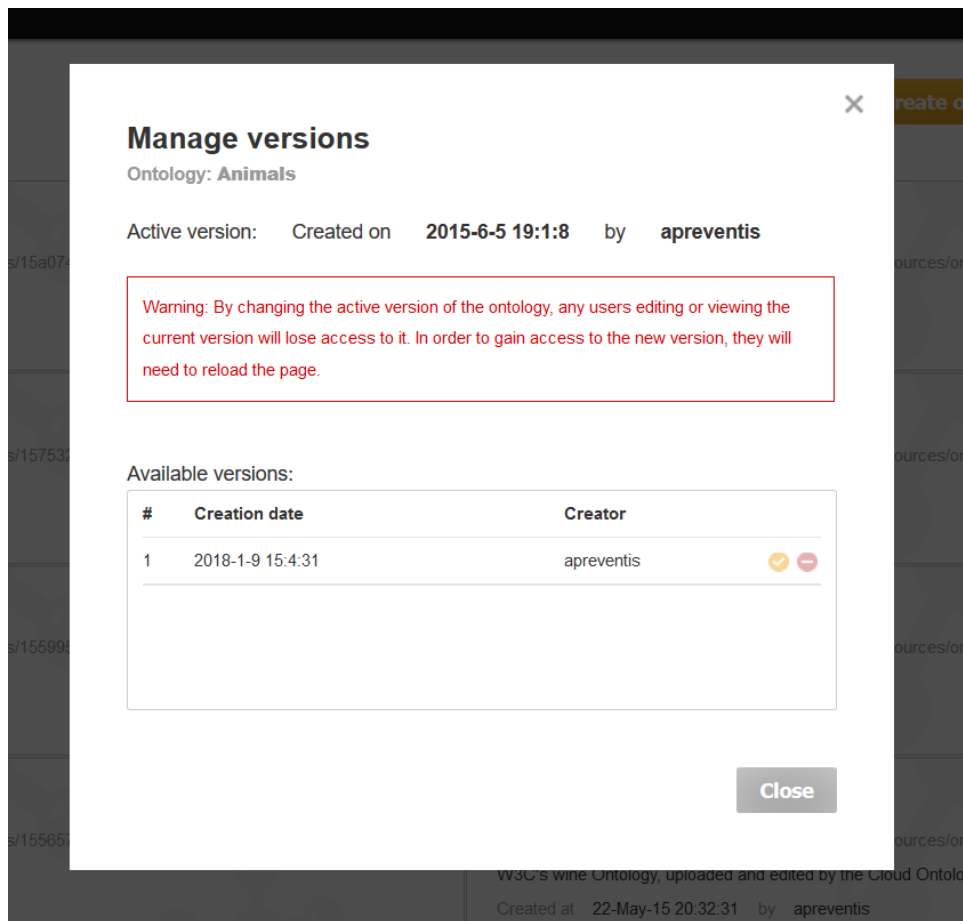


Figure 5.10: Manage ontology versions.

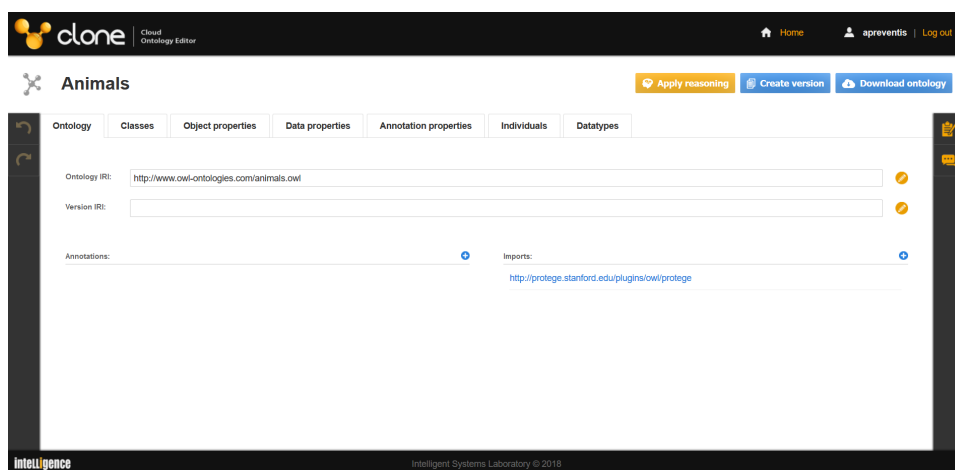


Figure 5.11: The Ontology Editor.



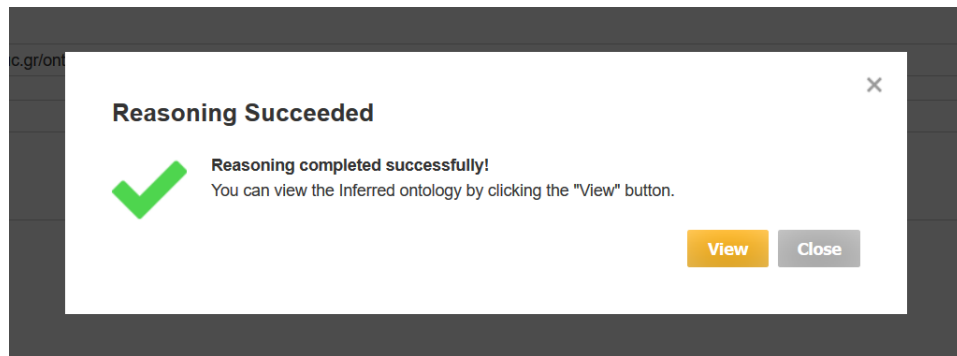


Figure 5.12: The modal that informs the user about the reasoning result.

If the user selects to apply reasoning on the ontology, CLONE will invoke the Reasoning Service to perform consistency checking. The user will be notified about the results in a dialog that will appear as soon as reasoning is complete (Figure 5.12). The user can then choose to view the inferred ontology in a separate window, or return to the edited ontology.

In the case that the users selects to create a version of the ontology, it is created instantly and added to the available versions that can be used as restoration points, as described in Subsection 5.2.2. The user is notified and he/she can continue editing the current version.

The last action, will download the ontology file of the current version in the same way that is described in Subsection 5.2.2.

- (ii) **The left vertical menu**, is the dark-colored column that is placed on the left of the page, as displayed in Figure 5.11. This menu contains two buttons, the “Undo” button that is no the top and the “Redo” that is below the former. Similarly to most text editing applications, CLONE allows to undo any change made to the ontology, by simply pressing “Undo”. The button is enabled after the user performs his/her first change and is able to revert up to 50 changes. The “Redo” button can be used to apply any change that has been previously “undone” using the “Undo” button.

Although the functionality of the buttons resembles this of a text-editor, they do not work in the same way. In CLONE, any changes are immediately saved in the ontology and made visible to the other users. They are even logged to the change history. In order to implement the “undo” and “redo” functionality, for every action that is performed, a *reverse action* —an action that reverts the previously made change— is stored in a stack. Whenever the user selects to undo an action, the last stored reverse action is executed, reverting the previous user-

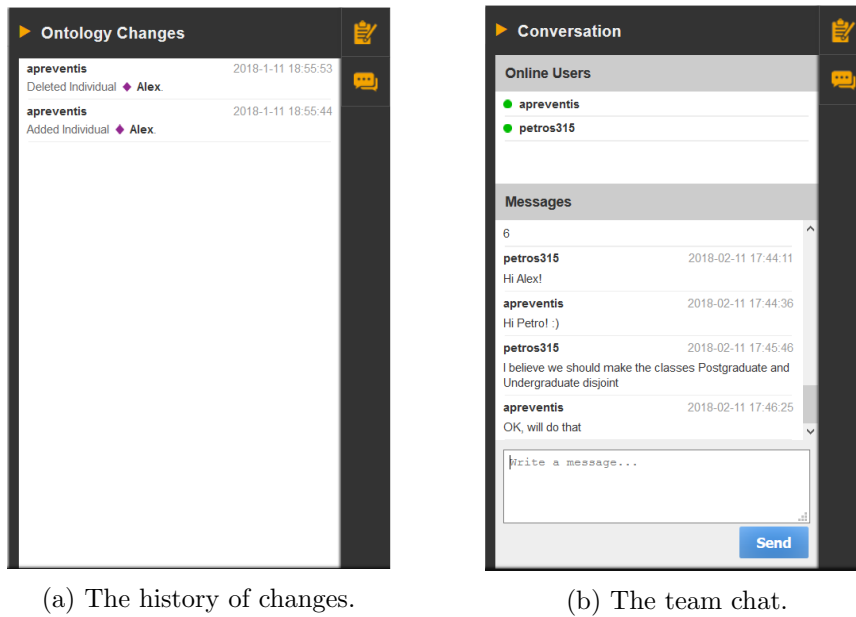


Figure 5.13: The ontology change history panel and the team conversation panel.

made action. For instance, suppose that a user wants to add a new Individual in the ontology. The moment that the individual is added, an action that will remove it from the ontology is stored in the undo stack. When the user selects to undo the last change, the action is retrieved from the stack and executed, removing the individual. The “Redo” stack works similarly.

- (iii) **The right vertical menu**, is the dark-colored column that is placed on the right of the page. It contains two buttons that give access to two of the most essential features of CLONE. On the top of the menu is placed the “Change History” button and below that, is the “Conversation” button. When clicking on each of these buttons, a panel will expand showing the history of changes that took place on the ontology, or the team conversation on the ontology. Figure 5.13 displays both these panels.

**The Change History** panel is where all the changes that have taken place on the ontology are displayed. Whenever a change is applied on the ontology or its entities is created, updated or deleted a new history record is inserted in the change-log database. A history record keeps track of (a) the user that performed the action, (b) a description of this action and the date and (c) time that it took place.

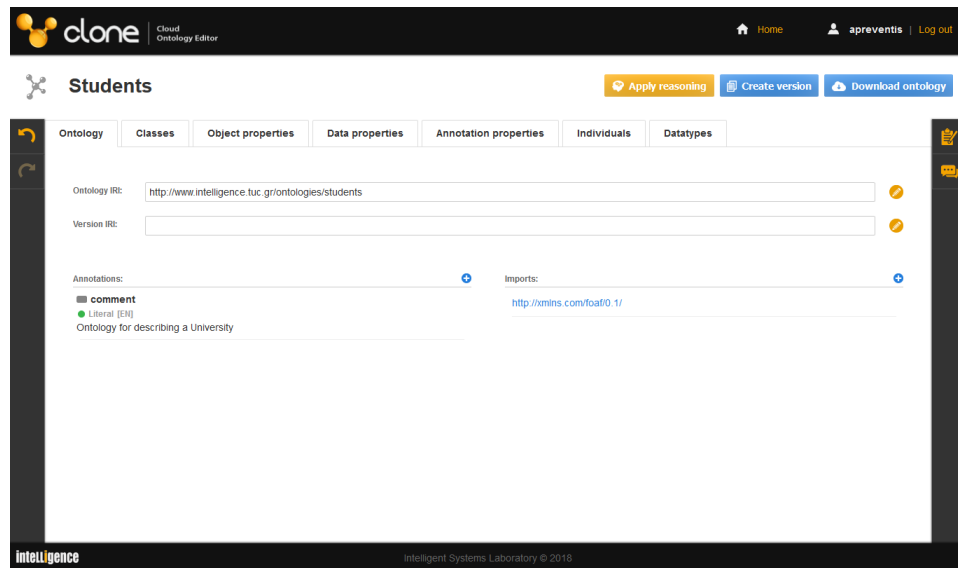


Figure 5.14: The ontology tab gives access to the basic properties of an ontology.

**The Ontology Conversation** panel allows for real time communication between the development team of an ontology. It facilitates knowledge sharing and exchange of ideas, while browsing the ontology or between its entities. It also informs about other users that are currently online. Conversation messages are visible to all users in the development team, even if they are not online at the time of sending.

- (iv) **The central panel**, is the main panel of CLONE. It contains 7 tabs, each one providing access to different type of ontology entities. These tabs are described in the following Subsection.

## 5.2.4 Ontology Tabs

### 5.2.4.1 The Ontology tab

This tab gives access to specific ontology properties. These are (a) the *Ontology IRI* which is used to uniquely identify an ontology, (b) the *Version IRI* that is used as an optional unique identifier for the ontology's version, (c) *Annotations* that can be used to associate information to the ontology—for example to add a short description, or the names of the creators—and (d) *Imports* that allow specifying other Ontology IRIs in order to gain access to their entities. Figure 5.14 illustrates the Ontology tab, along with the properties that can be accessed.

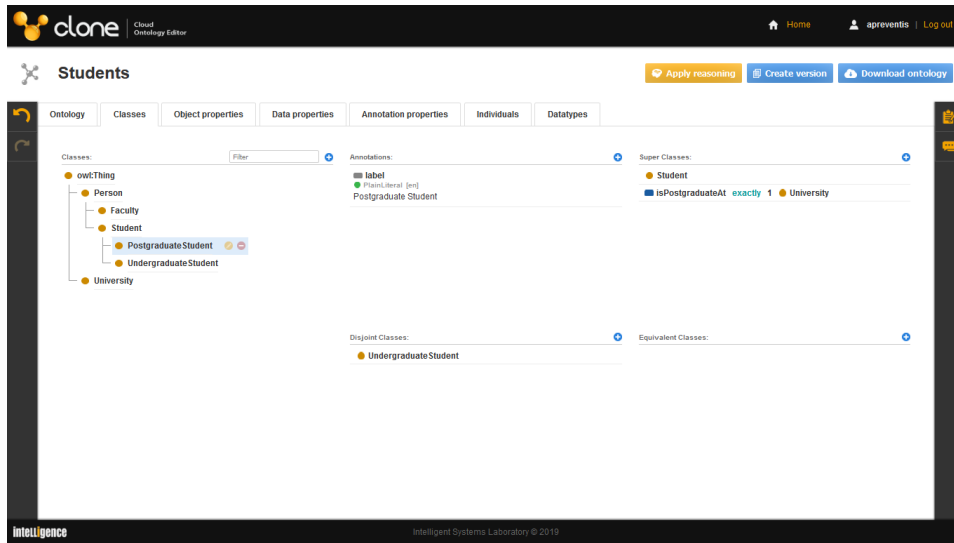


Figure 5.15: The classes tab enables managing ontology classes.

#### 5.2.4.2 Classes

Classes provide an abstraction mechanism for grouping resources with similar characteristics. Every OWL class is associated with a set of individuals, that is called the class extension.

This tab allows users to browse through the classes that have been defined in an ontology. It also enables users create new classes and edit or delete existing ones. The tab is divided into 5 panels that are described below.

The classes panel enlists all the ontology classes. The classes are presented in a tree view. Each class will appear below its parent. In the case that a class has been set to have more than one super-classes (that do not belong in the same sub-tree) it will appear under both. By moving the mouse over a class, two buttons appear next to its name, one for editing the its name (and URI) and one for removing the it from the ontology. By clicking on a class in the list, its related classes will be displayed in the other 4 panels.

The Annotations panel is used for displaying annotations that have been added on the selected class. Clicking the “plus” allows adding new annotations, using the *Annotation Editor*. Existing annotations can be removed by clicking the “minus” button that appears when moving the mouse over them.

The Super Classes panel displays the super-classes of the selected class. These can either be named classes or anonymous<sup>2</sup>. New super-classes can

<sup>2</sup>An anonymous class is described by the the restrictions that are placed on the class extension, whereas named classes are described by their URI.

be added using the *Class Expression Editor*.

Similarly to the Super Classes tab, the Disjoint Classes tab is used to display the classes that have been defined to be disjoint with the selected and the Equivalent Classes tab, these that have been defined as equivalent. These classes also can be named or anonymous. To add any disjoint or equivalent classes, the user can click on the “plus” button on the respective tab, that will pop-up the *Class Expression Editor*.

#### 5.2.4.3 Object Properties

OWL Object properties represent relations between two Individuals. This tab allows users to define new object properties, edit existing ones or remove them from the ontology definition.

The Object Properties tab displays the object properties that have been defined in an ontology, as well as their characteristics and associations. Users can browse through existing project properties, create new, edit or delete existing ones. This tab is divided in 8 panels, that are described below.

The *Object Properties* panel enlists the object properties that are included in the ontology. As in the case of classes, the properties are displayed in hierarchical structure. This panel provides functionality to create/edit/remove object properties. Users can use the search input on the top of the list to filter the properties by name.

Similarly to classes, there is the annotation panel that displays annotations on the selected object properties. Through this panel users can manage existing annotations or create new ones.

The *Characteristics* panel allows users to assign specific axioms<sup>3</sup> on the selected object property. Depending on their axioms, object properties can be characterized as:

- Functional Properties,
- Inverse-Functional Properties,
- Reflexive Properties,
- Irreflexive Properties,
- Symmetric Properties,
- Asymmetric Properties,
- Transitive Properties.

Users can define the selected object property *Domains* and *Ranges* classes from the respective panels. The Domains and Ranges can either be OWL named classes or Class Expressions.

---

<sup>3</sup>[https://www.w3.org/TR/owl2-syntax/#Object\\_Property\\_Axioms](https://www.w3.org/TR/owl2-syntax/#Object_Property_Axioms)

Also the following property relations can be specified on the selected object property, using the respective panels.

1. **Super Properties** are analogous to super classes. A sub-property inherits the axioms of its super-properties.

For example, consider the following axioms:

- (i) John *hasDog* Fluffy. (ii) *hasDog* *sub-class of* *hasPet*.

The ontology entails the following assertion: John *hasPet* Fluffy

2. **Equivalent Properties** are properties that are semantically equivalent to the selected property.

For example, consider the following axioms:

- (i) *hasBrother* *EquivalentTo* *hasMaleSibling*. (ii) George *hasBrother* Philip.

The ontology entails the following assertion: George *hasMaleSibling* Philip

3. **Disjoint Properties** that are properties that are pairwise disjoint to the selected property.

For example, consider the following axioms:

- (i) *hasMother* *disjointTo* *hasFather*. (ii) George *hasFather* John. (iii) George *hasMother* Mary.

At this point, the ontology is consistent and the disjoint axiom is satisfied. If one were to add the following assertion: “George *hasMother* John” the axiom would be invalidated and the ontology would become inconsistent.

It is important to note that in all three aforementioned panels, users can either add named object properties or Property Expressions.

#### 5.2.4.4 Data Properties

OWL *Data properties* are used for connecting individuals with literals. The Data Properties tab allows users manage the data property definitions in the ontology. The tab is identical to the Object Properties tab in all but two things:

- (i) The range of data properties is a *Datatype* whereas in object properties it is a Class Expression
- (ii) Data properties can have only one characteristic, the *Functional Property*.

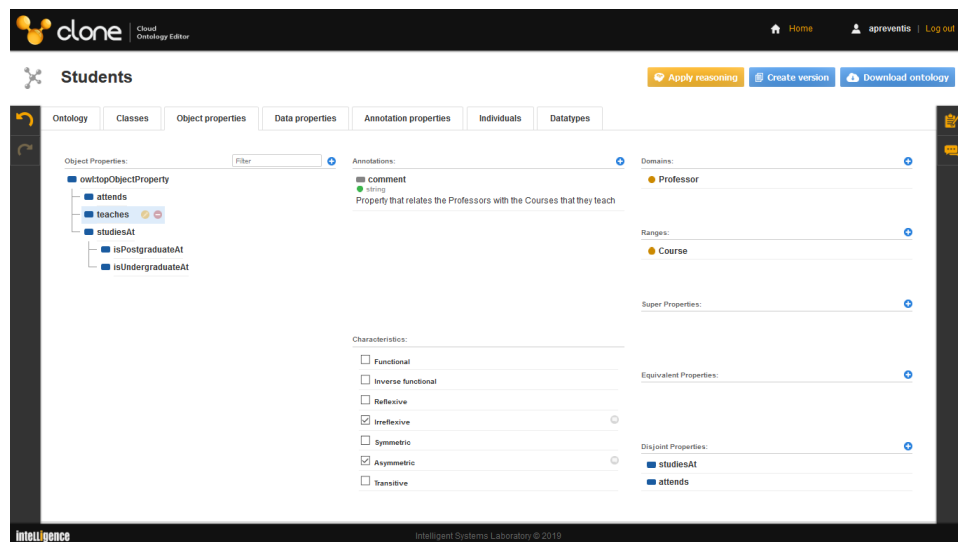


Figure 5.16: The Object Properties tab.

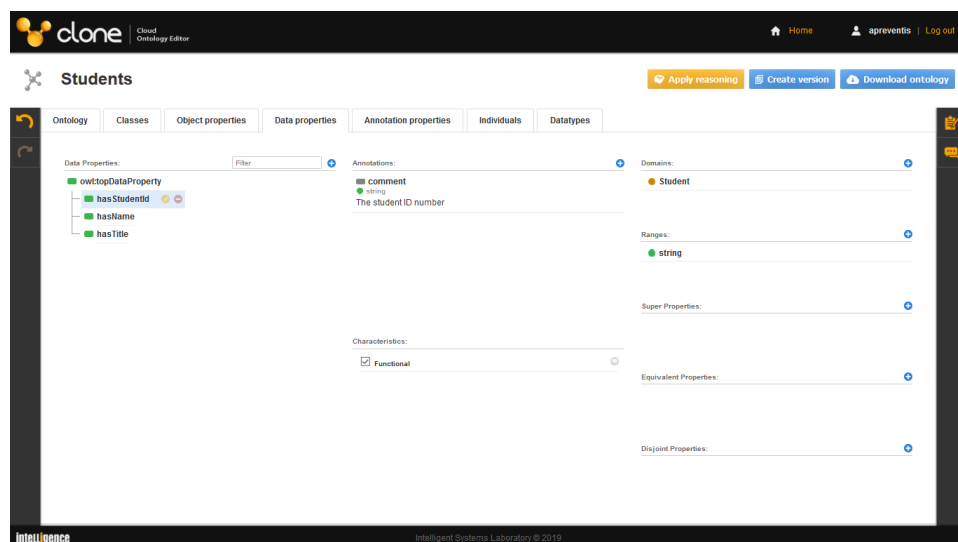


Figure 5.17: The Data Properties tab.

#### 5.2.4.5 Annotation Properties

*Annotation properties* can be used to provide an annotation for an ontology, axiom, or an IRI. For instance, one can add a comment on an object property using the annotation property “comment”, as displayed in Figure 5.16. Annotation properties (like other types of properties) can have Domains, Ranges and super-properties, all of which can be specified in the Annotation Properties tab. The Domains and Ranges can be specified using an entity’s IRI, thus any OWL entity can be set as an annotation property domain or range.

OWL allows for adding annotations on annotation properties as well, which can be accomplished via the annotations panel.

#### 5.2.4.6 Individuals

*Individuals* represent actual object of the domain of interest. They can be distinguished in *Named*, which are the individuals that have been assigned an explicit name and can be used in any ontology, and *Anonymous Individuals* that do not have a global name, thus they can only be referred in the ontology that they are contained. Clone’s current implementation supports Named Individuals.

Individuals can have one or more types, which are the OWL classes the individual belongs to. The types of individuals can be either anonymous or named. Clone allows for assigning both types on individuals, through the Types panel.

OWL is based on the *Open World Assumption*, meaning that nothing can be assumed that exists, unless it is explicitly stated. In the case of Individuals, this means that we cannot assume that two individuals are neither the same nor different unless it has been stated. These kind of statements can be added in the *Same Individuals* and *Different Individuals* panels, respectively.

As we have already mentioned, object properties represent the relationships between two individuals. After having selected the individual to associate (the *subject*) from the *Individuals* list, the user can use the *Object Property Assertions* panel to associate it with another individual (the *object*) via an object property (the *predicate*).

Individuals can also be associated with specific values, that can vary between strings, integers, decimals, floats and others. These associations can be done from the *Data Property Assertions* panel. Similarly to Object Property Assertions, the user can select the Individual that they want to associate with a value (the subject) from the Individuals list, and then select the *Datatype* and type-in the value.

Finally from this tab, users can specify *Negative Object Property Assertions* and *Negative Data Property Assertions*. These are object and data



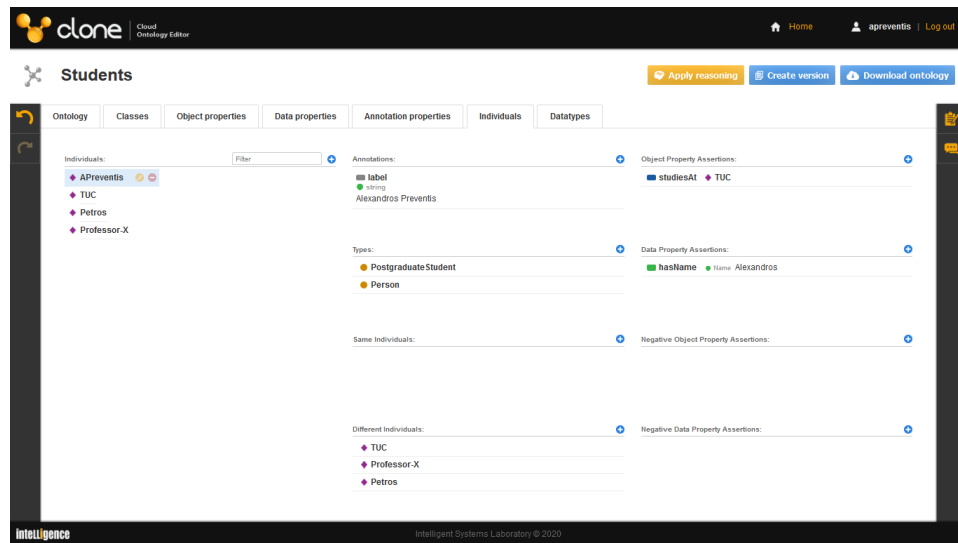


Figure 5.18: The Individuals tab.

property assertions that an individual cannot have, and they are introduced to similarly to object and data property assertions respectively.

#### 5.2.4.7 Datatypes

*Datatypes* define the particular values a data item can get. This tab includes a list with the built-in owl datatypes, but users can create their own datatypes as well. Users are also able to add annotations on datatypes from the “Annotations” panel. Finally they can add datatype definitions, which are used for defining that a datatype is semantically equivalent to another built-in datatype.

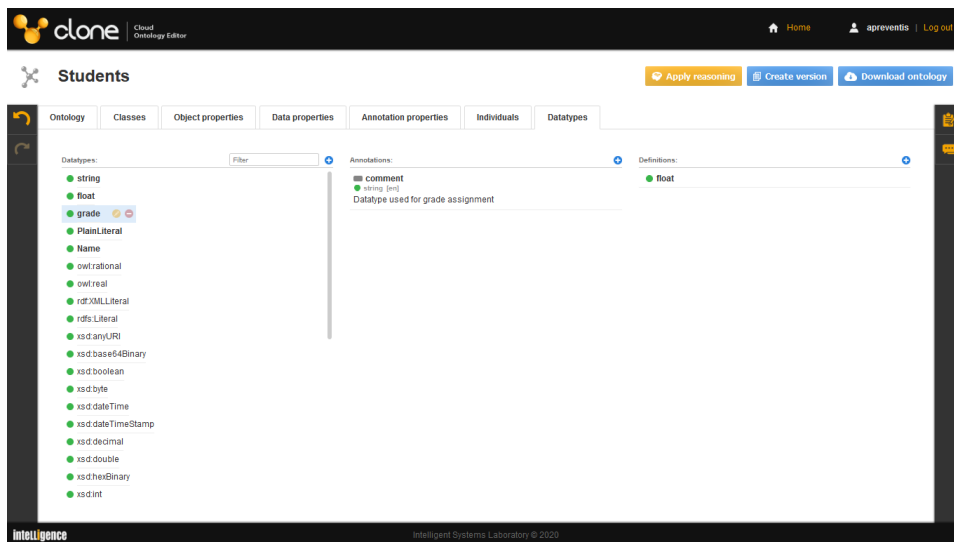


Figure 5.19: The Datatypes tab.

## Chapter 6

# Conclusions and Future Work

In this work we introduced CLONE (Cloud Ontology Editor), a Web-based ontology editor that provides real-time collaboration on designing and building RDF and OWL ontologies. CLONE offers a user-friendly environment that facilitates users take advantage of OWL's expressiveness to its full extent, without having specific knowledge on the representation language. The editor provides all the essential features of an ontology editor and also significant collaboration features, such as ontology versioning, cloud storage, various access levels, simultaneous editing, change history and many more.

CLONE has a component-based, service oriented architectural design. Each of its components operates as a service that exposes a REST API, making the system modular and allowing for extensibility, scalability and load-balancing.

In the short term, we plan to evaluate CLONE by running a usability study based on questionnaires addressing various aspects of system functionality (e.g. performance, functionality, ease of use) for different user categories.

We are currently working on extending CLONE's functionality to support more ontology formats such as Manchester Syntax and Turtle and also incorporate a triple store database for storing native ontology relations. This will allow handling large ontologies (that contain millions of triples) and also provide greater efficiency and data handling capabilities.

In the long-term we plan to support temporal ontologies, by integrating solutions such as Chronos [23], that add temporal aspects in ontologies and provide the means of describing relations that change over time.

# Appendices

# Appendix A

## Database Schemata

### A.1 Clone Core Component's SQL Database

**users:** Stores the following information on the registered users of the application.

first_name:	The user's first name.
last_name:	The user's last name.
email:	The user's email address. Unique column.
<u>username</u> :	The username of the user. Primary key.
password:	The user's password.

**ontologies:** Stores information on ontologies.

<u>id</u> :	Unique identifier of the ontology. Primary key.
name:	The name of the ontology.
<u>publisher</u> :	The username of the user who created the ontology. Foreign key from the <i>users</i> table.
description:	Ontology description.
location:	The location where the file of the ontology is stored.
creation_time:	The timestamp when the ontology was created.

**ontology\_viewers:** Stores information on the users that currently use an ontology.

<u>username</u> :	The username of the ontology user. Part of the primary key. Foreign key from the <i>users</i> table.
<u>ontology_uri</u> :	The ontology location. Part of the primary key.
last_ping:	The timestamp when the user notified the server that they are active.

**messages:** Stores the messages of the ontology conversations.

id: Unique identifier of the message. Primary key.

ontology: The ontology ID of the ontology where this message was sent. Foreign key from the *ontologies* table.

sender: The username of the user that sent the message. Foreign key from the *users* table.

body: The text of the message.

time\_sent: The datetime when the message was sent.

**roles:** Stores the user roles of the application.

id: Unique identifier of the role. Primary key.

type: A textual description of the role.

**permissions:** Stores the specific roles that have been assigned to each user of an ontology.

user: The username of the ontology user. Part of the primary key. Foreign key from the *users* table.

ontology: The ID of the ontology where the user role applies. Part of the primary key. Foreign key from the *ontologies* table.

role: The ID of the role that has been assigned to the user, in the particular ontology. Foreign key from the *roles* table.

**errors:** Stores the error code and messages used by the application.

code: Unique code that represents an error.

reason: The reason that has caused this error.

message: The error message that will be displayed to the user.

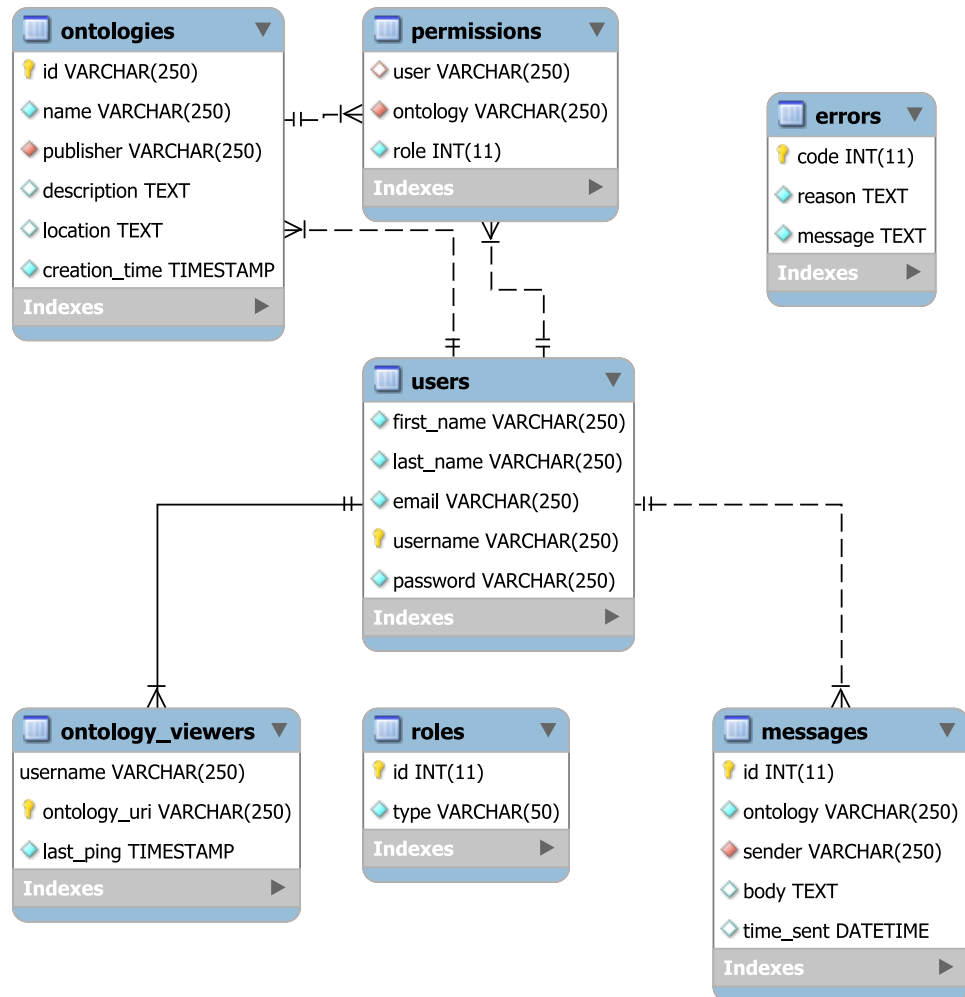


Figure A.1: Entity-Relationship diagram of Clone Core Component's database schema.

# Bibliography

- [1] S. Braun, A. Schmidt, A. Walter, G. Nagypal, and V. Zacharias, “Ontology Maturing: a Collaborative Web 2.0 Approach to Ontology Engineering,” in *Proceedings of the Workshop on Social and Collaborative Construction of Structured Knowledge at the 16th International World Wide Web Conference (WWW 07), Banff, Canada, 2007*.
- [2] T. Berners-Lee, J. Hendler, O. Lassila, *et al.*, “The semantic web,” *Scientific american*, vol. 284, no. 5, pp. 28–37, 2001.
- [3] G. Klyne and J. J. Carroll, “Resource Description Framework (RDF): Concepts and Abstract Syntax,” *W3C Recommendation*.
- [4] D. L. McGuinness, F. Van Harmelen, *et al.*, “Owl web ontology language overview,” *W3C recommendation*, vol. 10, 2004.
- [5] A. Miles and J. R. Pérez-Agüera, “Skos: Simple knowledge organisation for the web,” *Cataloging & Classification Quarterly*, vol. 43, no. 3-4, pp. 69–83, 2007.
- [6] M. Kifer, “Rule interchange format: The framework,” in *Web Reasoning and Rule Systems* (D. Calvanese and G. Lausen, eds.), vol. 5341 of *Lecture Notes in Computer Science*, pp. 1–11, Springer Berlin Heidelberg, 2008.
- [7] E. Prud’Hommeaux, A. Seaborne, *et al.*, “Sparql query language for rdf,” *W3C recommendation*, vol. 15, 2008.
- [8] B. McBride, “The resource description framework (rdf) and its vocabulary description language rdfs,” in *Handbook on Ontologies* (S. Staab and R. Studer, eds.), International Handbooks on Information Systems, pp. 51–65, Springer Berlin Heidelberg, 2004.
- [9] I. Horrocks, “Daml+ oil: a reason-able web ontology language,” in *Advances in Database Technology—EDBT 2002*, pp. 2–13, Springer, 2002.



- [10] B. Motik, P. F. Patel-Schneider, B. Parsia, C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, *et al.*, “Owl 2 web ontology language: Structural specification and functional-style syntax,” *W3C recommendation*, vol. 27, no. 65, p. 159, 2009.
- [11] M. Horridge and S. Bechhofer, “The owl api: A java api for owl ontologies,” *Semantic Web*, vol. 2, no. 1, 2011.
- [12] O. Corcho, M. Fernández-López, and A. Gómez-Pérez, “Ontological engineering: Principles, methods, tools and languages,” in *Ontologies for Software Engineering and Software Technology* (C. Calero, F. Ruiz, and M. Piattini, eds.), pp. 1–48, Springer Berlin Heidelberg, 2006.
- [13] J. Davies, Y. Sure, D. Vrandečić, S. Pinto, C. Tempich, and Y. Sure, “The diligent knowledge processes,” *Journal of Knowledge Management*, vol. 9, no. 5, pp. 85–96, 2005.
- [14] C. Tempich, E. Simperl, M. Luczak, R. Studer, and H. S. Pinto, “Argumentation-based ontology engineering,” *IEEE Intelligent Systems*, vol. 22, no. 6, pp. 52–59, 2007.
- [15] T. Tudorache, N. Noy, S. Tu, and M. Musen, “Supporting collaborative ontology development in protégé,” in *The Semantic Web - ISWC 2008* (A. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. Finin, and K. Thirunarayan, eds.), vol. 5318 of *Lecture Notes in Computer Science*, pp. 17–32, Springer Berlin Heidelberg, 2008.
- [16] M. Weiten, “Ontostudio® as a ontology engineering environment,” in *Semantic Knowledge Management* (J. Davies, M. Grobelnik, and D. Mladenić, eds.), pp. 51–60, Springer Berlin Heidelberg, 2009.
- [17] A. Kalyanpur, B. Parsia, E. Sirin, B. C. Grau, and J. Hendler, “Swoop: A web ontology editing browser,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 4, no. 2, pp. 144 – 153, 2006. Semantic Grid –The Convergence of Technologies.
- [18] E. Simperl and M. Luczak-Rösch, “Collaborative ontology engineering: a survey,” *The Knowledge Engineering Review*, vol. 29, no. 01, pp. 101–131, 2014.
- [19] S. Tramp, P. Frischmuth, and N. Heino, “Ontowiki—a semantic data wiki enabling the collaborative creation and (linked data) publication of rdf knowledge bases,” *Demo Proceedings of the EKAW*, vol. 2010, 2010.
- [20] P. Haase, H. Lewen, R. Studer, D. T. Tran, M. Erdmann, M. d’Aquin, and E. Motta, “The neon ontology engineering toolkit,” *WWW*, 2008.

- [21] T. Tudorache, C. Nyulas, N. F. Noy, and M. A. Musen, “WebProtégé: A Collaborative Ontology Editor and Knowledge Acquisition Tool for the Web,” *Semantic web*, vol. 4, pp. 89–99, Jan. 2013.
- [22] P. Warren, “Ontology users’ survey—summary of results,” *Month*, 2013.
- [23] A. Preventis, P. Marki, E. G. M. Petrakis, and S. Batsakis, “Chronos: A tool for handling temporal ontologies in protégé,” in *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, vol. 1, pp. 460–467, Nov 2012.